REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
14-05-2004	Final Report	15 June, 2002 - 14 May, 2004

4. TITLE AND SUBTITLE
Toward development of a virtual distributed control system

5a. CONTRACT NUMBER
N00014-02-1-0787

5b. GRANT NUMBER
N00014-02-1-0787

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Principal author: Moshe Kam
Additional authors: Andres Lebaudy, Yifeng Xu, Erwei Lin, Mianyu Wang

5d. PROJECT NUMBER
02PR13057-00

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Data Fusion Lab, Department of Electrical and Computer Engineering
Drexel University, 3141 Chestnut Street, Philadelphia, PA 19104
Subcontractor: Fairmount Automation, Inc.
4621 West Chester Pike, Newtown Square, PA 19073

8. PERFORMING ORGANIZATION REPORT NUMBER
Final-Phase2-v.001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
OFFICE OF NAVAL RESEARCH
BALLSTON CENTRE TOWER ONE
800 NORTH QUINCY STREET
ARLINGTON, VA 22217-5660

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited distribution

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We describe two simulation and design tools, developed for large-scale distributed control applications. At present, these applications require extensive experimentation with hardware, which is costly, slow, and inflexible. The new simulators allow computerized evaluation of communication requirements and discovery of potential bottlenecks in large-scale systems of actuators, sensors, transport lines and communication networks. They provide designers with easy, quick, and inexpensive methodology to examine, rate, and compare proposed designs. The first tool, the Communication Simulator, is used in early phases of the design, for quick but somewhat rough comparison of architectures. It requires estimates of the basic communication parameters of the nodes. The second tool, the Network Simulator, simulates node dynamics and communication traffic simultaneously, and provides detailed and accurate estimates of parameters such as packet loss, bandwidth utilization, channel throughput, and channel capacity. The simulators were validated using a large-scale dynamic architecture, the Chilled Water Reduced-Scale Advanced Demonstrator.

15. SUBJECT TERMS

Simulation, Automation, Distributed Control, Communication Networks, Control Network Protocols, ANSI/EIA-709, LONWORKS, FieldBus, Chilled Water Reduced-Scale Advanced Demonstrator.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Moshe Kam
UNC	UNC	UNC	SAR	158	19b. TELEPHONE NUMBER *(Include area code)* (215) 895-6920

TABLE OF CONTENTS

TABLE OF CONTENTS 1

LIST OF FIGURES 3

LIST OF TABLES 5

LIST OF COMMONLY USED ACRONYMS 6

1. INTRODUCTION 7

1.1 PURPOSE AND SCOPE 7
1.2 ORGANIZATION OF THE DOCUMENT 8
1.3 OBJECTIVES AND ACHIEVEMENTS 9

2. DEVELOPMENT REVIEW (SUMMARY) 10

3. SIMULATION GENERATOR 12

3.1 SIMULATION GENERATOR ARCHITECTURE 12
3.2 NETWORK DEVICE APPLICATION TRANSLATION 14
3.3 NETWORK CONFIGURATION TOOL 15
3.4 ADDRESS TABLE AND ROUTER FORWARDING TABLES 17
3.5 NETWORK VARIABLE CONFIGURATION AND FORMAT FILES 17
3.6 CONTROL PLANT SIMULATION CONFIGURATION 18
3.7 USING THE SIMULATION GENERATOR: A STEP-BY-STEP EXAMPLE 19

4. NETWORK SIMULATOR 29

4.1 NETWORK SIMULATOR ARCHITECTURE 29
4.2 CHANNEL MODEL 30
4.3 CHANNEL TYPES 32
4.4 ETHERNET MODEL 33
4.5 NETWORK DAMAGE 34
4.6 MULTI-CHANNEL AND ROUTER 36
4.7 SIMULATION THREADS, TIME AND SYNCHRONIZATION 41
4.8 NETWORK DEVICE APPLICATION 45
4.9 CONTROL PLANT APPLICATION 47
4.10 DEVICE AND CHANNEL STATISTICS 49
4.11 INTEGRATING THE NETWORK SIMULATOR WITH THIRD-PARTY SIMULATION TOOLS: A PRACTICAL EXAMPLE 50

5. DESIGN TOOL USING THE NETWORK SIMULATOR 54

5.1 SOFTWARE ARCHITECTURE 54
5.2 GUI AND APPLICATION INTEGRATION 55
5.3 ANALYSIS TOOLS 57
5.4 DEVICE APPLICATION DEBUGGING 58

6. VDCS VALIDATION 60

7. THE COMMUNICATION SIMULATOR 62

7.1 BACKGROUND - OPNET MODELER DESCRIPTION 62
7.2 IMPLEMENTATION OF LONWORKS NETWORKS MODELING TOOL 65
7.3 ANALYTICAL MODELS OF LONWORKS P-CSMA ALGORITHM 84
7.4 MODEL VALIDATION 90
7.5 CONCLUSION 96

8. CONCLUSION AND OUTLOOK 98

9. REFERENCES 99

APPENDIX I : DEVICE AND PLANT APPLICATION DEFINITION FOR CODE GENERATION AUTOMATION 100

APPENDIX II : DEVICE APPLICATION SIMULATION API 116

APPENDIX III : DESIGN TOOL USER GUIDE 124

1. INTRODUCTION 126
2. GETTING STARTED 129
3. DESIGNING A LONWORKS® NETWORK 130
4. SIMULATING A NETWORK 133
5. USING THE ANALYSIS TOOLS 138
6. DEBUGGING DEVICE APPLICATIONS 146
7. ADVANCED TOPICS 148

APPENDIX A 150

APPENDIX B 156

APPENDIX IV: LNS DATABASE 157

APPENDIX V: COM TECHNOLOGY 158

LIST OF FIGURES

FIGURE 1 SIMULATION GENERATOR 12
FIGURE 2 SIMULATION DEFINITION GENERATION 14
FIGURE 3 NEURON C SOURCE TRANSLATOR ARCHITECTURE 14
FIGURE 4 NETWORK CONFIGURATION TOOL 16
FIGURE 5 OPEN LONMAKER TO DESIGN NETWORK 19
FIGURE 6 PROGRAM DEVICE APPLICATION PROGRAM IN LONBUILDER 20
FIGURE 7 DESIGN THE RESISTOR FAN NETWORK USING LONMAKER 21
FIGURE 8 PROGRAM THE PLANT SIMULATION CODE 22
FIGURE 9 LONWORK SIMULATION GENERATOR MAIN DIALOG 23
FIGURE 10 OPEN NETWORK DATABASE DIALOG 24
FIGURE 11 PLANT SIMULATION DIALOG 25
FIGURE 12 PLANT INTERFACE DIALOG 26
FIGURE 13 DEVICES AND PLANT INTERFACE CONNECTION DIALOG 27
FIGURE 14 SIMULATION GENERATION PROGRESS DIALOG 27
FIGURE 15 COMPLICATION OF THE SIMULATION 28
FIGURE 16 NETWORK SIMULATOR 29
FIGURE 17 NETWORK SIMULATION COMPONENT 30
FIGURE 18 NETWORK SIMULATION ARCHITECTURE 30
FIGURE 19 MESSAGE PROPAGATION ALONG A CHANNE 31
FIGURE 20 PACKET PROPAGATION THROUGH A CHANNEL 32
FIGURE 21 ETHERNET AND LONTALK CHANNELS 33
FIGURE 22 CHANNEL DAMAGE SCENARIO 34
FIGURE 23 PACKET PROPAGATION IN A DAMAGED CHANNEL 35
FIGURE 24 ROUTER STRUCTURE 36
FIGURE 25 SOFTWARE ARCHITECTURE OF ROUTER 37
FIGURE 26 ROUTING ALGORITHM 39
FIGURE 27 SEARCHING THROUGH A NETWORK TO SET THE FORWARDING TABLES 40
FIGURE 28 ETHERNET ROUTER OBJECT 40
FIGURE 29 NETWORK SIMULATION THREADS 41
FIGURE 30 DEVICE APPLICATION IMPLEMENTATION 45
FIGURE 31 PLANT AND NETWORK SIMULATION MODEL 47
FIGURE 32 NETWORK SIMULATION INTERFACE 48
FIGURE 33 SYSTEM SIMULATION ARCHTECTURE 49
FIGURE 34 AN ILLUSTRATION OF THE PHYSICAL PIPING SYSTEM WITH SMART VALVES 51
FIGURE 35 PIPING SYSTEM AND CONTROL NETWORK SCHEMA 51
FIGURE 36 ASNE DEMO SNAPSHOT 52
FIGURE 37 APPLICATION MONITOR WITH GRAPHIC 53
FIGURE 38 NETWORK SIMULATION TOOL ARCHITECTURE IN DESIGN TOOL 54
FIGURE 39 SIMULATION GENERATOR ARCHITECTURE IN DESIGN TOOL 55
FIGURE 40 SOFTWARE ARCHITECTURE AND DEPENDENCY 56
FIGURE 41 DESIGN TOOL FOLDER HIERARCHY 56
FIGURE 42 MESSAGE PROPAGATION WITH MEMORY PIPES. 57
FIGURE 43 RSAD NETWORK DESIGN IN LONMAKER 60
FIGURE 44 HIERARCHICAL ORGANIZATION OF EDITORS 64
FIGURE 45 FINITE STATE MACHINE OF THE LONTALK MAC LAYER 67
FIGURE 46 GUI OF LONWORKS COMMUNICATION SIMUALTOR 68
FIGURE 47 HIERARCHICAL STRUCTURE OF NETWORK MODEL 71
FIGURE 48 DATA TRANSMISSION BETWEEN TWO NODES 72
FIGURE 49 LAYERED STRUCTURE OF LONWORKS DEVICE NODE MODEL 72
FIGURE 50 VARIOUS MAC ATTRIBUTES 73
FIGURE 51 PROCESS MODEL OF *SOURCE* MODULE 75
FIGURE 52 PROCESS MODEL OF *LON_MAC_INF* INTERFACE MODULE 76
FIGURE 53 PROCESS MODEL OF *LON_MAC* MODULE 77

FIGURE 54 PROCESS MODEL OF *SINK* MODULE 77
FIGURE 55 LAYERED STRUCTURE OF LONMANAGER PROTOCOL ANALYZER 78
FIGURE 56 FSM OF LPA MODEL 78
FIGURE 57 FRAME FORMAT OF MAC LAYER PROTOCOL DATA UNIT 80
FIGURE 58 FRAME FORMAT FROM OPNET 80
FIGURE 59 ICI ATTRIBUTES OF LON_MAC_REQUEST 81
FIGURE 60 ICI ATTRIBUTES OF LON_MAC_INF 81
FIGURE 61 COLLECTABLE STATISTICS IN LONWORKS MODELING TOOL 83
FIGURE 62 MARKOV CHAIN MODEL FOR THE BACKLOG WINDOW SIZE 86
FIGURE 63 COLLISION PROBABILITY OF CHANNEL VERSUS. NUMBER OF NODES 91
FIGURE 64 COLLISION PROBABILITY OF NODE VERSUS. NUMBER OF NODES 92
FIGURE 65 CHANNEL THROUGHPUTS VERSUS. TRAFFIC OFFERED BY EACH NODE (6 NODES, UNACK) 93
FIGURE 66 BANDWIDTH UTILIZATION VERSUS. TRAFFIC OFFERED BY EACH NODE (6 NODES, UNACK) 93
FIGURE 67 CHANNEL THROUGHPUTS VERSUS. TRAFFIC OFFERED BY EACH NODE (8 NODES, UNACK) 94
FIGURE 68 BANDWIDTH UTILIZATION VERSUS. TRAFFIC OFFERED BY EACH NODE (6 NODES, UNACK) 94
FIGURE 69 CHANNEL BACKLOG (UNACK SERVICE) 95
FIGURE 70 CHANNEL BACKLOG (ACK SERVICE) 96
FIGURE 71 COUNTING OF COLLISION (UNACK SERVICE) 97
FIGURE 72 END-TO-END DELAY (UNACK SERVICE) 97
FIGURE 73 TRANSMISSION ATTEMPTS (UNACK SERVICE) 97
FIGURE 74 ADDITIONAL STATISTICS (UNACK SERVICE) 97

LIST OF TABLES

TABLE 1 NETWORK APPLICATION DEFINITION FILES 15
TABLE 2 NETWORK CONFIGURATION FILES 16
TABLE 3 SIMULATION CONTROLLER SYNCHRONIZATION ALGORITHM 44
TABLE 4 THREAD SYNCHRONIZATION ALGORITHM 44
TABLE 5 COMPARISON OF NETWORK TRAFFIC STATISTICS 61
TABLE 6 THE LIST OF PROCESS MODELS 74
TABLE 7 PACKET FORMAT 79
TABLE 8 ICI FOMRAT 80
TABLE 9 STATISTICS STORED IN VECTOR OUTPUT FILE 82
TABLE 10 STATISTICS STORED IN SCALAR OUTPUT FILE FOR LONWORKS NODE DEVICE 82

LIST OF COMMONLY USED ACRONYMS

API	Application Program Interface – a set of routines, protocols and tools for building software applications.
COM	Component Object Model (Appendix V)
DLL	Dynamic Link Library – a collection of short programs, called by other, often larger, programs to execute specific tasks.
FIFO	First in/First out
ICI	Interface control information
ID	Identity, identification
IDE	Integrated Development Environment
LAN	Local Area Network
LNS	LONWORKS Network Service (Appendix IV)
LPA	LonManager Protocol Analyzer
MAC	Medium Access Control (a MAC layer manages and maintains communication between stations; it moves data packets to and from one network interface card to another across a shared channel.)
MPDU	MAC sublayer Protocol Data Unit
MS VC++	Microsoft Visual C++ (C++ = the computer language C++)
NC	Neuron C
NV	Network Variable(s)
OPNET	Optimized Network Engineering Tool – a modeling library (section 7.1 and [11])
OSI	Open Source Initiative (see www.opensource.com)
SNVT	Standard Network Variable Types, see www.lonmark.org/products/snvtfile.htm
VDCS	Virtual Distributed Control System

1. INTRODUCTION

1.1 Purpose and scope

This document is the final report of the project "Toward development of a virtual distributed control system (Phase II)". The report describes an 18-month effort to develop a simulation and visualization tool that will allow comprehensive analysis of shipboard distributed control networks.

We describe the simulation tool that emerged from the effort. The tool is designed to assist engineers who build distributed architectures for coordinated control of multiple devices and subsystems. While not limited to any application type, our designs target shipboard distributed control applications that use LONWORKS.

In order to design a distributed control architecture, engineers typically develop goals expressed as performance indices, and draw block diagrams of potential architectures. These potential architectures require specification of communication and interconnection networks and listings of sensors and actuators, including their locations and types (*e.g.*, PID, bang-bang). Next, design tools such as LonMaker and NodeBuilder are engaged to assist in realization.

In the absence of analytical and simulation tools, each potential architecture needs to be realized and tested "live" with the actual hardware. This approach is expensive, slow, inflexible, and may miss critical limitations if certain modes are not excited during experimentation. The objective of this project was to provide a computerized simulation tool, which would allow inexpensive automated comparison of design alternatives and analysis of different designs without physical implementation. The general input used by our tool is the block diagrams of the proposed system, and the source code planned for the sensory and control devices. The user also supplies: the duration of the planned simulation, parameters of the desired communication service (*e.g.*, acknowledge receipt of packets), and sampling rates. The output is live and historical statistics of the communication infrastructure (*e.g.*, packet loss, bandwidth utilization, channel throughput, and channel capacity).

Using our software, users can assess future performance of the design's communication infrastructure, identity bottlenecks and contingencies, and estimate available bandwidths and throughputs. This assessment is provided at two levels of abstractions: the first (Communication Simulator) is intended for preliminary quick block diagram design; the second (Network Simulator) is intended for detailed simulation of a fully instrumented network.

1.2 Organization of the document

Chapter 1 provides definition of scope and purpose, as well as a summary of goals and achievements.

Chapter 2 provides a development review.

Chapter 3 and 4 provide a detailed technical description of the Network Simulator.

Chapter 5 describes the Network Simulator computerized design tool (for which a User Manual is provided in Appendix III).

Chapter 6 describes validation experiments.

Chapter 7 describes the Communication Simulator.

Chapter 8 states goals for Phase III of the project.

Validations of the various tools developed in the course of this project are available in the following chapters: Chapter 3 - section 3.7, resistor/fan test platform; Chapter 6, Chilled Water Reduced-scale Advanced Demonstrator; Chapter 7 – section 7.4, validation of communication parameters.

Auxiliary materials and additional explanations were gathered in the appendices. Appendix III is the User Manual for the Network Simulator.

1.3 Objectives and achievements

Component-level distributed control systems are comprised of sensing and computation nodes, distributed throughout a plant (such as a ship) and connected peer-to-peer through device-level control networks. These decentralized architectures hold promise in large-scale shipboard automation, because they often exhibit increased survivability as well as graceful degradation in the face of component failure. However, decentralization by itself does not guarantee survivability, and new impediments such as communication delays, increased processing overhead, and unexpected interdependencies may even reduce survivability for some decentralized control architectures. The main objective of this project was to develop a tool that would predict how decentralized control systems perform, and would allow designers and users to evaluate and compare competing implementations. We seek an automated analysis, simulation, and visualization device that incorporates the dynamics of the control network into the simulation model and provides realistic prediction of performance and tradeoffs. The device should achieve these goals at a reasonable investment in hardware and computation power.

The objective of Phase I of this effort (2001-2) was to demonstrate feasibility of this approach, through the development of a functional, reduced-capability simulation tool. The tool was to simulate a distributed control system consisting of a small number of nodes (up to 10) that share a common communication channel, and to validate this simulation with a physical test plant. We focused our effort on the *LonTalk* protocol, pioneered by the Echelon Corporation. We chose this protocol because it is gaining acceptance within the Navy research community as the protocol of choice for Naval applications.

The objective of Phase II was to increase the dimension of the plants that can be simulated to about 100, and to provide a tool that Navy engineers can use when considering distributed control architectures. The effort resulted in two automated tools: a Communication Simulator and a Network Simulator. The *Communication Simulator* (Chapter 7) simulates a network at a high level of abstraction. It assumes that communication requirements for sensors and actuators are known, and proceeds to simulate the resulting communication network and identity the main operational parameters. The Communication Simulator is quick, but it requires significant prior information. It is suitable for the initial phases of the design, for demonstration of feasibility, and for basic comparisons of competing architectures.

The *Network Simulator* (Chapter 4) is suitable for the second phase of the design, when the sensors and actuators were selected, and overall performance of the fully-instrumented system is sought. It runs, in parallel, the sensor/actuator suite and the communication infrastructure. The Network Simulator requires less prior information, but is slower and more complex when compared with the Communications Simulator. It is suitable for performance prediction of the final design and for sensitivity studies, but less suitable for quick comparisons of basic block diagrams.

2. DEVELOPMENT REVIEW (Summary)

In phase I of the VDCS project, we developed and validated a software tool to simulate small-scale distributed control systems that communicate using the *LonTalk* protocol. The tool incorporated the dynamics of the underlying communications network into a composite simulation. We applied the tool to study problems consisting of less than ten distributed control nodes.

In phase II of the project, our primary objective was to provide a fully automated software tool to handle medium-scale distributed control systems (i.e., approximately 100 computing nodes). To this end, we developed processing and visualization modules for architectures with large numbers of nodes. This objective required the development of automated tools that convert LONWORKS device-level programs, network configuration data, and plant simulation code into a form that allows automated and reliable performance prediction and analysis.

Phase II of the project resulted in two major products: a Network Simulator and a Communication Simulator. Both simulators assume stable operation of local controllers and provide users with a detailed simulation and prediction of communication performance of the distributed control architecture. The differences between the simulators are in the level of abstraction.

The Network Simulator (Chapter 4) uses source code (written in Neuron C) for the control devices in order to determine the communication parameters, and proceeds to simulate the emerging communication scheme as the controllers operate. The Communication Simulator (Chapter 5) requires prior knowledge of the communication parameters and does not run the control algorithm during the simulation. As a result it is faster than the network simulator, though of courses it requires much more information. In the design process, the Communication Simulator should be used first, with estimated parameters for the planned devices in the various nodes. Once satisfactory performance was established, the Network Simulator should be used, with the detailed design of the devices at each node.

In addition to the two simulators, the following key milestones were achieved:

2.1 We developed tools to extract network configuration information automatically and directly from a LONWORKS Network Services (LNS) database. This database is created by the LonMaker utility commonly used by engineers to configure a LONWORKS system. Engineers use the database to specify and store the communication network architecture, and mathematical definitions of components are extracted and installed in the architecture in order to facilitate simulation and testing. By interpreting the information in the database, a user would only need to enter information once and will not need to learn how to use additional design tools. We developed an LNS extraction tool that retrieves the network topology and configuration information directly from LNS database. (see Appendix IV)

2.2 We developed a means to integrate third-party simulation and visualization tools with the VDCS platform (*e.g.*, Wonderware InTouch, Flowmaster, LabView). The VDCS platform provides a high-fidelity simulation of LonTalk network traffic.

2.3 We devised a means to introduce network failures into a VDCS simulation. Specifically, VDCS enables users to test system performance in the presence of (1) device failures, (2) channel failures, and/or (3) router failures. We developed a simple user-interface to allow an engineer to identify a device, router, or network link that is damaged while a simulation is in progress.

2.4 We added simulation support for routers and bridges[1]. As distributed control systems grow in size, design engineers often use routers and bridges to group computing nodes into sub-networks (each sub-network has its own channel). The election to segment communication traffic is not only a good design practice; it is often dictated by communication protocols that physically limit the number of nodes that can share the same network channel (in LONWORKS the limit is 64 nodes, in FairNET the limit is 32 nodes). The VDCS simulation platform supports both configured and learning [2]modes of LonTalk routers.

2.5 We developed improved analysis tools for user to assess the performance of designs. These analysis tools provide all of the performance data that can be obtained by using a protocol analyzer on a real-world system. Moreover, the tools provide several useful performance metrics that cannot be computed by any other means (protocol analyzers do not provide this information).

2.6 We developed a design tool to simulate network channel traffic and evaluate the network messaging schemes of a preliminary LONWORKS network design. The idea is to use this tool during the early stages of a distributed control network design process.

2.7 We incorporated Component Object Modeling (COM) technology into the VDCS platform. The primary benefit of using COM is that it provides a mechanism to create standardized interfaces between applications. This is useful to integrate VDCS with third-party simulation software for physical systems (*e.g.*, a fluid system). We designed a testing environment that supports both simulation generation and execution. This effort required restructuring of the VDCS software architecture by developing COM interfaces and components to isolate the user application from the network simulation and graphical user interface (GUI). (see Appendix V)

[1] A *bridge* between two sub-networks passes all communication traffic from each sub-network to the other 'blindly'. A *router* between sub-networks A and B is more discriminating: it passes message packets from sub-network A to sub-network B only if those packets are addressed to a node on sub-network B (or to a node on another sub-network connected, directly or indirectly, to sub-network B.)

[2] LonTalk routers can operate in one of two modes: (i) configured or (ii) learning. In *configured* mode, the router is fully informed about the overall network architecture. In the *learning* mode, the router is not informed about the network architecture and must develop a map of the network by tracking message packets. As it learns, the router may pass message packets from one sub-network to another unnecessarily, thereby increasing bandwidth consumption.

3. SIMULATION GENERATOR

In this chapter we describe a pre-processing device that prepares data for the Network Simulator (Chapter 4). The VDCS test platform separates the network simulation process into two steps: (1) preparing the network simulation to represent a specific network design to be simulated, and (2) performing the network simulation. Accordingly, VDCS includes a *Simulation Generator* and a *Network Simulator*. The Simulation Generator retrieves the network design from the LNS database and the device-level user applications. It then interprets the network definition and builds the network simulation files. Based on these files, the Network Simulator creates a network simulation and runs the simulation. This chapter describes the simulation generator architecture.

3.1 Simulation Generator Architecture

The Simulation Generator is an automation tool that creates a series of files that define a network simulation. It extracts the network definitions from the LNS database and stores these definitions in a number of simulation files that are used by other VDCS components. Some of the files are loaded directly by the Network Simulator to establish the simulation. The Simulation Generator feeds the other files to a third-party compiler that creates the device and plant simulation dynamic link libraries (DLLs). These DLLs are dynamically linked by the Network Simulator to execute the simulation.

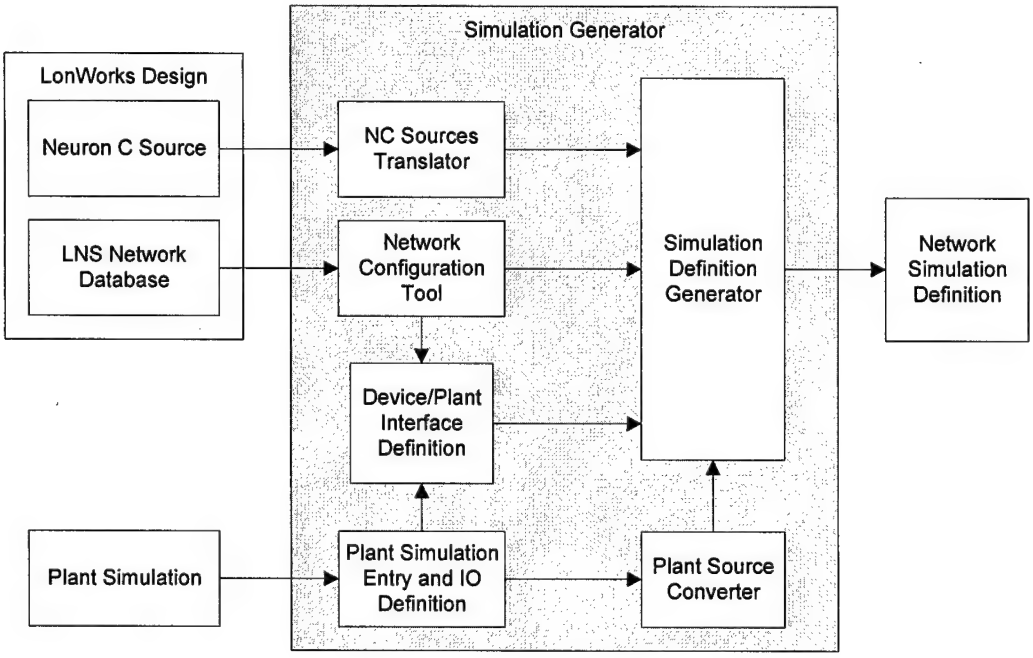


Figure 1 Simulation Generator

Figure 1 shows the functional block diagram of the Simulation Generator. Inputs to the Simulation Generator include the network design created in LonMaker and NodeBuilder[3], and a user-defined plant simulation file (Appendix I specifies the format for this file). The Simulation Generator

[3] Echelon Company, http://www.echelon.com/products/development/nodebuilder/default.htm

consists of a *Network Configuration Tool*, a *Neuron-C Source Translator*, a *Device/Plant Interface Definition Tool*, a *Plant Simulation Entry and I/O Definition Tool*, a *Plant Source Converter*, and a *Simulation Definition Generator*.

The Neuron C Source Translator interprets the Neuron C code of the user applications into an object-oriented C++ form that can be interpreted by any third party compiler. We developed an Application Program Interface (API) to support the most often-used Neuron C functions. Appendix III provides a template that the Neuron-C code must adhere to in order to be understood by the Source Translator. The output of the Source Translator is a set of files that represent the user application simulation code. These files are compiled by a third-party tool to create a user application simulation DLL module.

The Plant Source Converter converts the plant application code into a set of object-oriented C++ files. These files are used to build a plant application simulation DLL module. (The plant application code simulates the behavior of the physical systems and processes that the distributed control system is regulating.)

The Network Configuration Tool accesses the LNS database and extracts information about all of the available network configurations that the user has defined in LonMaker. Once the user chooses the specific network to be simulated, the tool explores the LNS database for the network topology and configuration information. The specific information gathered in this extraction process includes the network name, subsystem number and names, channel numbers, channel name, number of devices and routers, the device name, unique node ID, domain ID, subnet ID, node ID, application template, device network interface, router name, router priority property, router mode, NV definitions, bindings, etc. The information is stored in a simulation data structure referred by the Simulation Definition Generator and Device/Plant Interface Definition module.

The Plant Simulation Entry and I/O Definition component provides the user with the authority to specify the plant simulation file and to define the plant I/O signals. The Device/Plant Interface Definition provides an interface for the user to bind plant I/O signals with device I/O signals. The Simulation Definition Generator combines and processes the results from the other components to generate the simulation files and network configuration files. The architecture of the Simulation Definition Generator is shown in Figure 2.

VDCS uses the Microsoft Visual C++ (MS VC++) Compiler to build a device application COM component and a plant application COM component based on the Network Simulation Definition files. VDCS creates a MS VC++ project file and launches the compiler to build the target files. If the simulation modules are built successfully, the Simulation Generator will exit and the Network Simulator may begin to execute the simulation.

When building a network simulation, the Simulation Generator gathers the following network configuration information: the network topology, the channel and device configuration, the network connections, and the individual device applications. This information constitutes the network definition. To assist with this network definition, a network configuration tool was developed. It is available as an automated tool that generates the entire network simulation from the original design, referring to the LNS database, network application source, and plant simulation code.

The network design procedure using the simulation generator is explained below in detail, using an example, namely the *Resistor / Fan Test*[5]. The simulation generator is customized in our Design tool (Chapter 5); (A simplified version is integrated into the design tool software package).

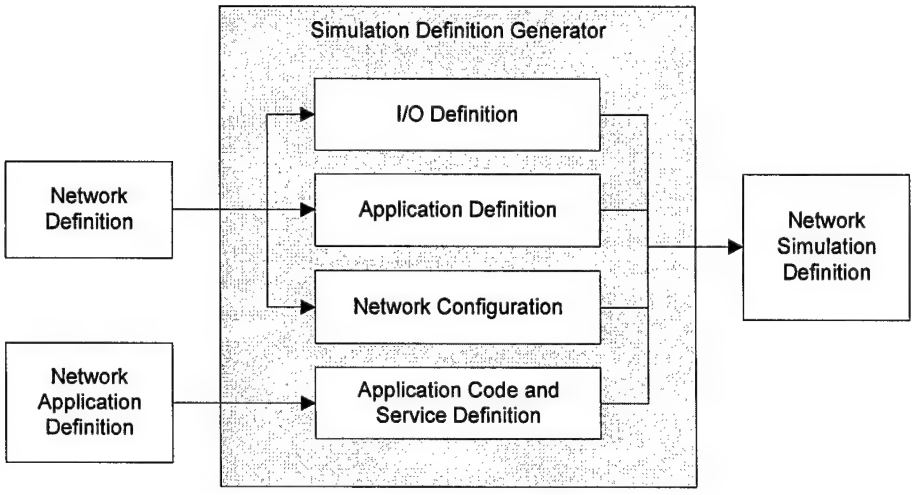


Figure 2 Simulation Definition Generation

3.2 Network Device Application Translation

The *Neuron C Source Translator* translates the device source code into the form specified by the device application definition. It also derives the network interface of a network device and the event list that will be checked by the scheduler. The Neuron C Source Translator has three function blocks as shown in Figure 3. These are Device Application Translation, Network Interface and Event Definition, and NV Format Definition.

The *Application Translation module* converts all the device application source code into a C++ format that complies with the VDCS device application definition. The application source translation attempts to maintain the characteristics of the Neuron C (NC) language [4], and similarity to the original code, to preserve code readability. However, functions and variables defined in the device application definition are created according to the NC source code. The device application definition is depicted in Appendix I.

The device source code is not in the LNS database. However, the directory of each device application .XIF file is recorded in the LNS database. The .XIF file is generated based on the device application source code and is usually stored in a path that has a fixed relation to the source code. Base on this fact, the source code is located in the Network Configuration Tool for translation.

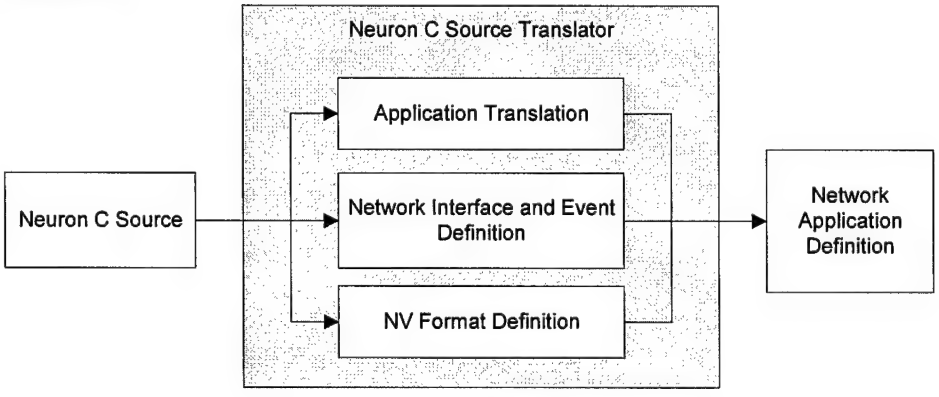


Figure 3 Neuron C Source Translator Architecture

The Network Interface and Event Definition module create two files: Interface.fmt and Event_NV.fmt. Interface.fmt holds the network interface information of each device; and Event_NV.fmt keeps the lists of network variable events of network device templates. The NV Format Definition builds the NV format APIs for the NV configuration tool according to the SNVT format file and user defined format file (See www.lonmark.org/products/snvtfile.htm). The network variable configuration is described in section 3.5. The resulting files from these functions make up the Network Application Definition, which are listed in Table 1.

Table 1 Network Application Definition Files

include.cpp	Included files of device applications
DeviceAppgm.h	Device application declaration
DeviceAppgm.cpp	Device application definition
Interface.fmt	Device interface
Event_NV.fmt	List of NV event of each device template
NVfmt.h	NV configuration API declaration
NVfmt.cpp	NV configuration API definition

Additionally, a special code format is formulated for the network service device application, so that the user can write an application code for the network service device in the defined format to simulate a network service device. The code may be developed in NC or C with the code mirroring the functions of the network service device. The definition of the network service device is recorded the same way as a regular device in the configuration files.

3.3 Network Configuration Tool

Most of the network configuration information is in the *LONWORKS Network Services (LNS) database*. This database is created by the LonMaker utility commonly used by engineers to configure a LONWORKS system. By interpreting the information in the database, a method to extract the network configuration information directly from a LNS database is developed. Network information retrieval is performed by setting the LNS Database to "offnet" mode. The retrieved network information is processed to form the network definition. Figure 4, shows the architecture and function blocks of the Network Configuration Tool.

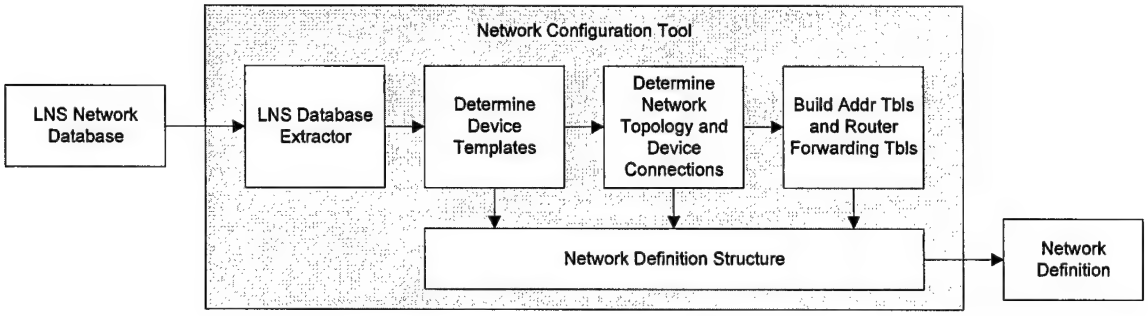


Figure 4 Network Configuration Tool

The LNS Database Extractor accesses the LNS database specified by the user to search for the network topology and configuration information. The information gathered in the extraction includes the channel numbers, channel name, number of devices and routers, the device name, unique node ID, subnet ID, node ID, application template, device network interface, router name, router priority property, router mode, NV definitions, and bindings. The information is stored in a simulation data structure used by other function blocks.

While the LNS extraction tool can access most configuration information from the LNS database, certain data can not be obtained directly. These include *device address tables* and *router forwarding tables*, and the *Neuron-C source files*. Fortunately, we are able to determine the location of the application source file based on the path of device application .XIF file as described in section 3.2. We also developed our own algorithms to generate these tables based on the network design rule and implications in the available network configurations. The software rebuilds these tables, including the router subnet and group forwarding tables. The definitions of these tables are provided in section 3.4.

The Network Configuration Tool produces the Network Definition that consists of the list of network configuration files shown in Table 2. During the network simulation initialization, the Network Simulator scans these files. The network simulation objects are created according to the `network.fmt` file. The configuration specified in `node.fmt` is implemented in each device application object, and the network variables are defined according to the `nv.fmt file`. The node IDs are loaded for each device in both the network simulation in the Network Simulator and the VDCS user applications. The program IDs are loaded in the user application to determine the appropriate applications for each network devices. The I/O connections are also configured at the same time according to `io.xls`. The I/O connection implementation is described in section 3.6.

Table 2 Network Configuration Files

File Name	Description
network.fmt	Network topology
node.fmt	Network device configurations
nv.fmt	Network variable definitions
id.fmt	Network device Ids
io.xls	IO configuration of the control plant and the network devices

3.4 Address Table and Router Forwarding Tables

The address table defines the network destination address for implicitly addressed messages and network variables of a network device. Each entry of the table represents an address. The maximum number of address table entries is 15. The definition of the address table can be found in *Motorola LONWORKS Technology Device Data book.* Since the configuration of the address table is not available from the LNS database, the address table is built based on implied network variable connections.

A message sent between two network devices is treated as a *subnet/node-addressing message.* The source and destination address may be obtained from the two devices containing the network variables. For subnet/node addressing, one address table entry is used in the source device to save the address of the destination device. Only one address table entry is used if there are one or more variable bindings between the two devices.

A message sent to more than one network device is considered a *group message.* Every device in a group will receive this message. The Group ID identifies this group uniquely. Any device in the group is given an index starting from 0. Each device in the group has an address table entry to identify its membership in the group by saving the addressing of the group and the device index.

A router uses forwarding tables to determine whether to forward an incoming message to the destination channel or drop it. Messages generally fall into two groups according to their addressing: subnet/node message or group message. Accordingly, there are two forwarding tables: subnet forwarding table and group forwarding table. The definition of the router tables and construction of these tables are explained in section 4.6.2.

3.5 Network Variable Configuration and Format Files

Occasionally the value of a network variable in a network device needs to be modified in order to make changes to the device application. The LonMaker Network Variable Browser provides this feature, called *NV configuration.* VDCS also provides a network variable configuration tool. This tool is able to check the current value of a network variable and modify it. It is necessary for the configuration tool to browse a network variable and correctly display the current value in the defined format.

To browse a network variable, the `interface.fmt` file is used as a reference to locate a network variable defined in a device. A hash table is built for looking up the data type of a network variable. The NV configuration tool first obtains the index to the network variable by checking the `interface.fmt` file, and then gets the network variable data type by looking up the hash table with the network variable index.

To display the network variable correctly according to its data type, the display format must be defined. Format files are used to determine these formats. A format file for the standard network variable type is defined and available in LonMaker. A modified version of the same file is used in VDCS. Meanwhile, a user-defined file may also be provided to support formats for user-defined data type. Functions are developed to read the network variable value and present it into strings in the defined format. These functions also convert a new value from user input based on the format and update the network variable.

3.6 Control Plant Simulation Configuration

The plant simulation configuration defines the plant application and the connections between the plant I/O and network device I/O. The plant simulation configuration includes four steps: plant application simulation entry, I/O definition, network/plant interface definition, and plant simulation translation.

Plant Simulation Entry

The plant application is assumed to be simulated in iterations that the internal states of a plant are changed from one iteration to the next. The plant simulation code specifies the one step simulation of the plant at a given sampling period. During this step, the user specifies the path of a plant simulation file, as well as the sampling period. The plant simulation information is stored in the network definition data structure. The definition of the plant application simulation file is described in Appendix I *Device and Plant Application Definition for Code Generation Automation.*

Plant I/O definition

A plant simulation has a set of state variables, some of which are used for input and output of the plant simulation. However the simulation file does not define the plant input and output. The configuration tool scans the plant simulation file and lists all the defined variables in the plant simulation code. A GUI is introduced to the configuration tool for the user to define the output and input variables.

Network/Plant Interface Definition

The network/plant interface defines the connection of the network hardware I/O and the plant I/O and determines the data transition. A connection tool is developed for the user to connect the I/Os. The simulation definition generator generates a connection configuration file `io.xls` containing the I/O definition and the connection definition. As described, the network simulation interface functions as a buffer that connects both the device I/O and plant I/O to exchange data.

Plant Simulation Translation

The last step is to build the simulation project. In building the plant simulation project, the plant simulation file is translated into a format that complies with the plant simulation definition and put into file `PlantAppgm.h` and `PlanAppgm.cpp` file. The I/O definition specifies the connection of the plant I/O and the network simulation interfaces buffers. The I/O definition for the plant simulation part is embedded into the application code. The I/O definition of network devices is similarly defined in device simulation. The I/O API functions are presented to encapsulate the implementation of the I/O connections.

3.7 Using the Simulation Generator: A Step-By-Step Example

The resistor/fan physical test platform was developed in Phase I of the VDCS Project to validate the tool. In Phase I, we manually translated all of the Neuron-C code and manually re-created the network configuration in a form that VDCS could understand. With the availability of the simulation generation tools developed under Phase II, we can now perform this step automatically. We provide below a step-by-step guide to using the automated tools to generate a simulation of the resistor-fan system. A detailed description of the simulated fan-resistor system is provided in Chapter 4 of the Phase I final report [7].

There are three major steps to build a network simulation. The first step is to design the network using a network design tool like LonMaker. The second step is to design the plant simulation. The third step is to build the simulation using the simulation code generator. After generating the simulation code, the network simulation is able to run using the simulation platform. Each step is described in detail below.

Step 1. Design the Resistor_Fan Network.

(1) LonMaker was used to design the *Resistor_Fan* network as shown in Figure 5. We set the network to be *offnet*, since there is no hardware attached.

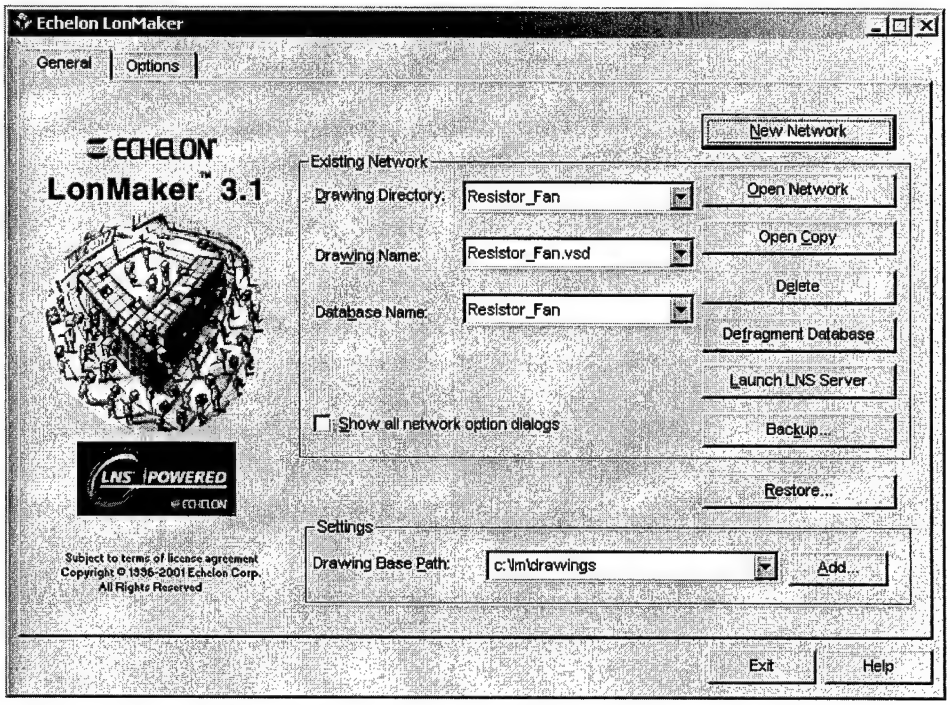


Figure 5 Open LonMaker to Design Network

(2) Define the Network Device Applications. Program the Neuron C device applications for the temperature module and the fan control module in LonBuilder. Figure 6 shows the Neuron C code that implemented the bang-bang controllers used to regulate the fan-resistor network.

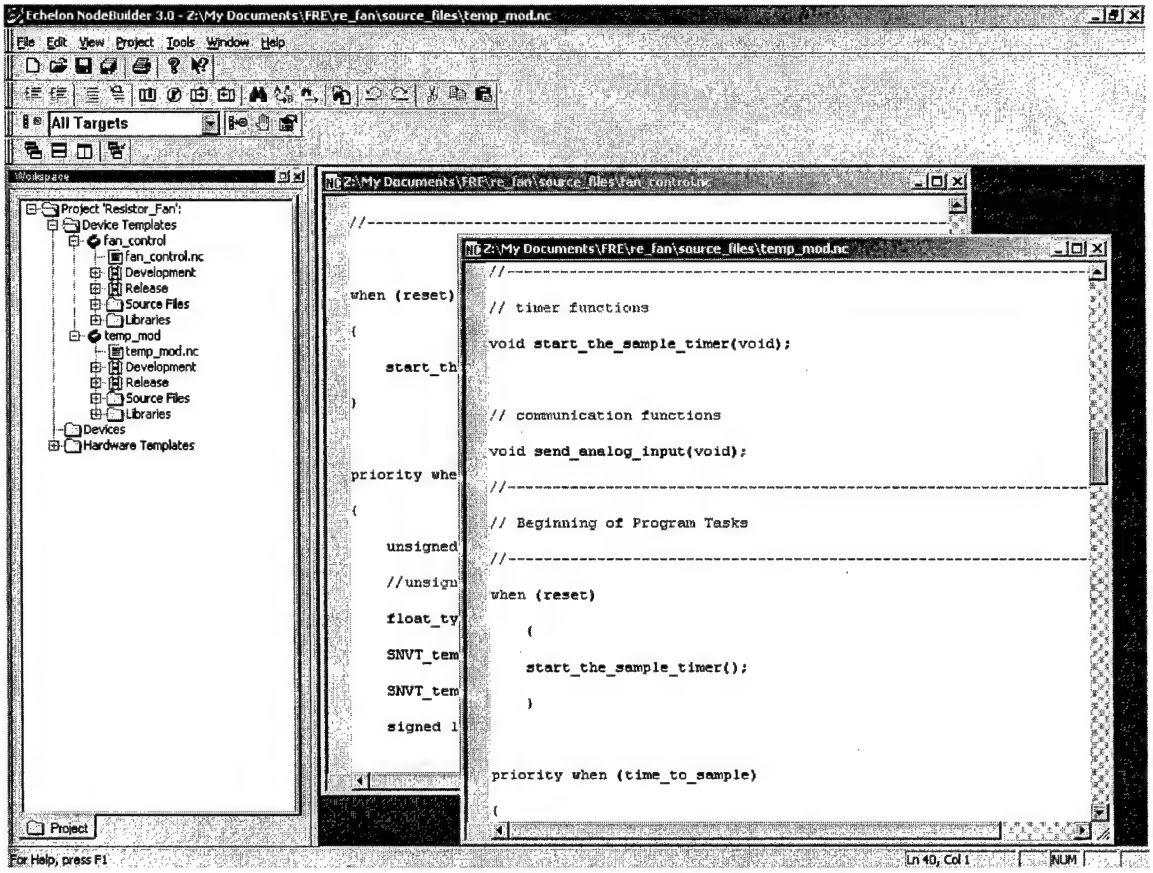


Figure 6 Program Device Application Program in LonBuilder

(3) Use LonMaker to design the Resistor_Fan network, define network variables, and bind network variables, as shown in Figure 7.

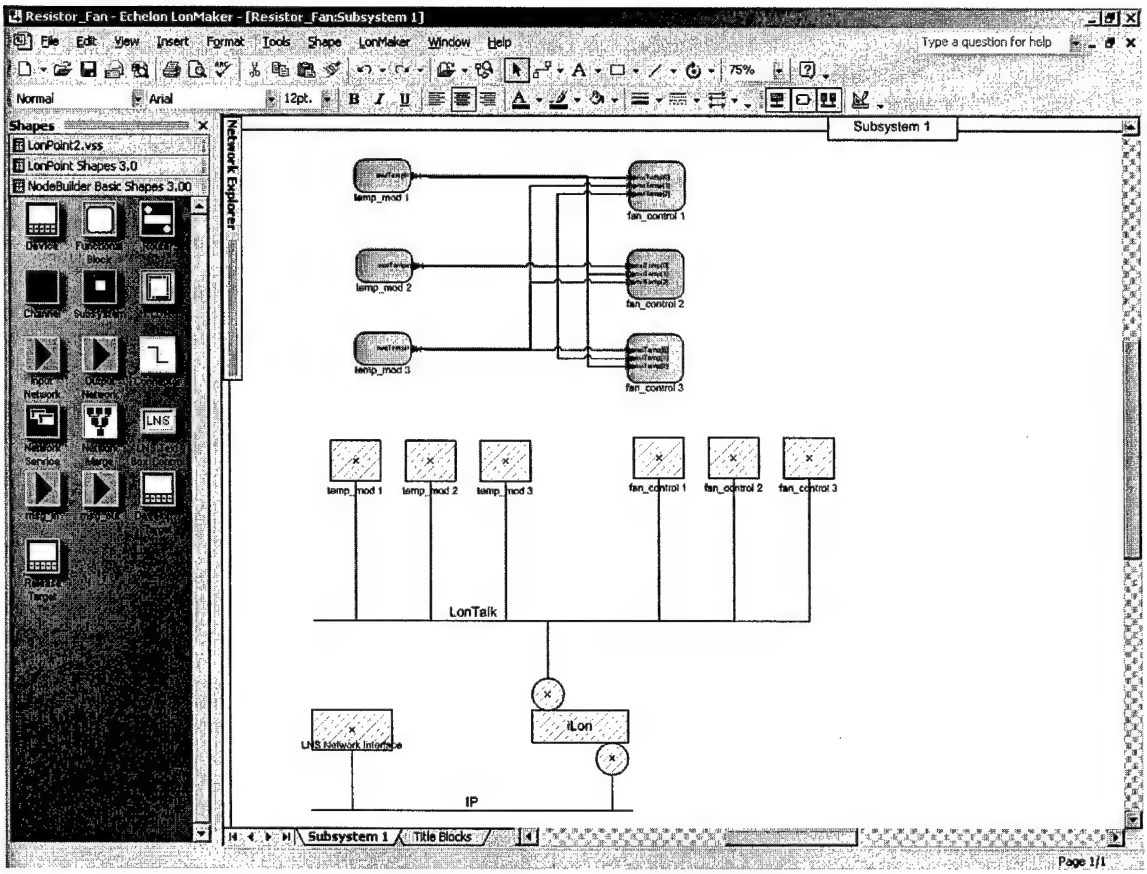


Figure 7 Design the Resistor Fan Network Using LonMaker

Step 2. Design the Plant Simulation.

Design the plant simulation based on the plant physical model. Program the plant simulation using any program editor. In Figure 8 the Crimson Editor is used to program the application. The plant simulation code should adhere to the format defined in Appendix I. The main() (see www.crimsoneditor.com) function should execute one step of the plant simulation code. Subroutines are allowed, but no special header files should be included.

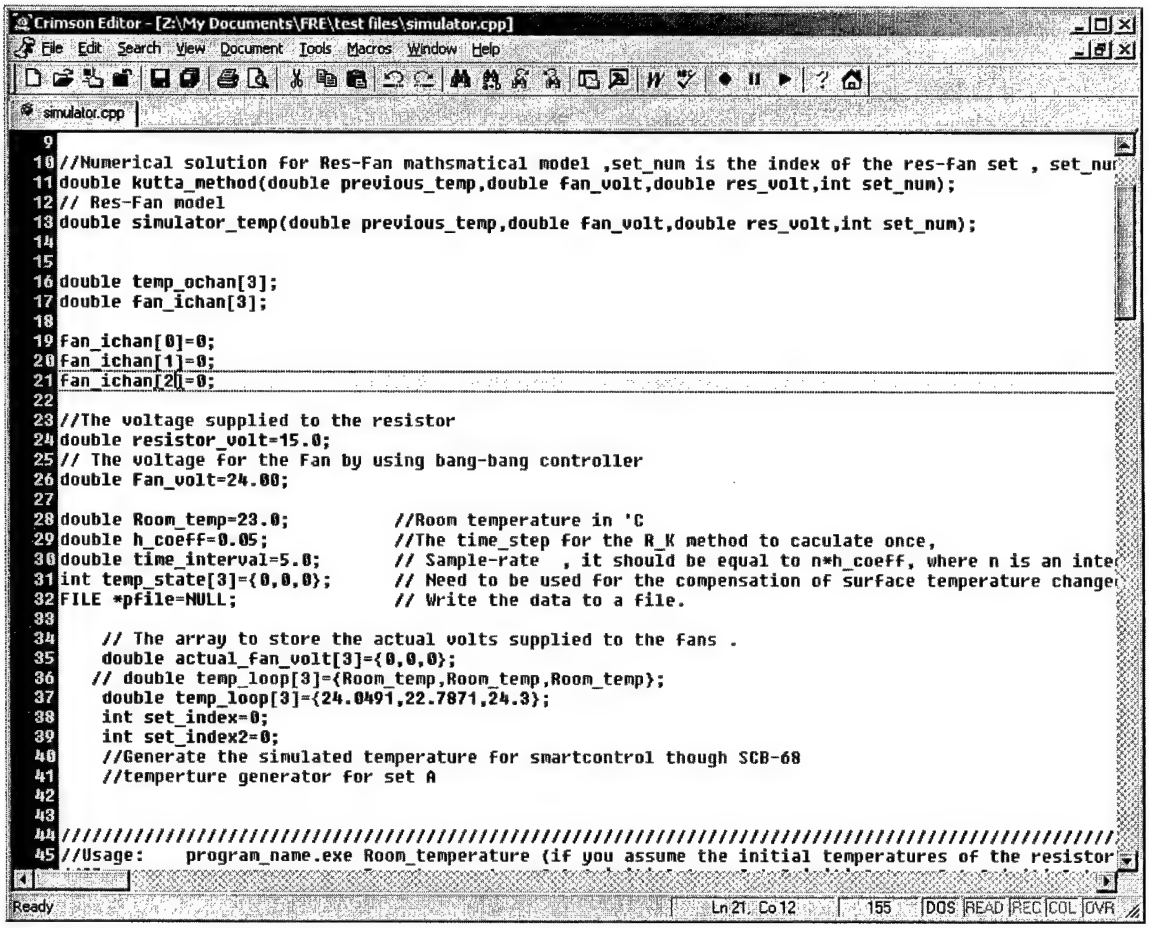


Figure 8 Program the Plant Simulation Code

Step 3. Build Simulation using LonWork Network Simulation Generator.

There are three steps to build the simulation code using the simulation generator. The first step is LNS database extraction. The second step is plant simulation definition. The third step is network and plant interface definition. After these three steps are completed, the generator will build the simulation code. To begin, open the simulation generator; a dialog as shown in Figure 9 will be displayed.

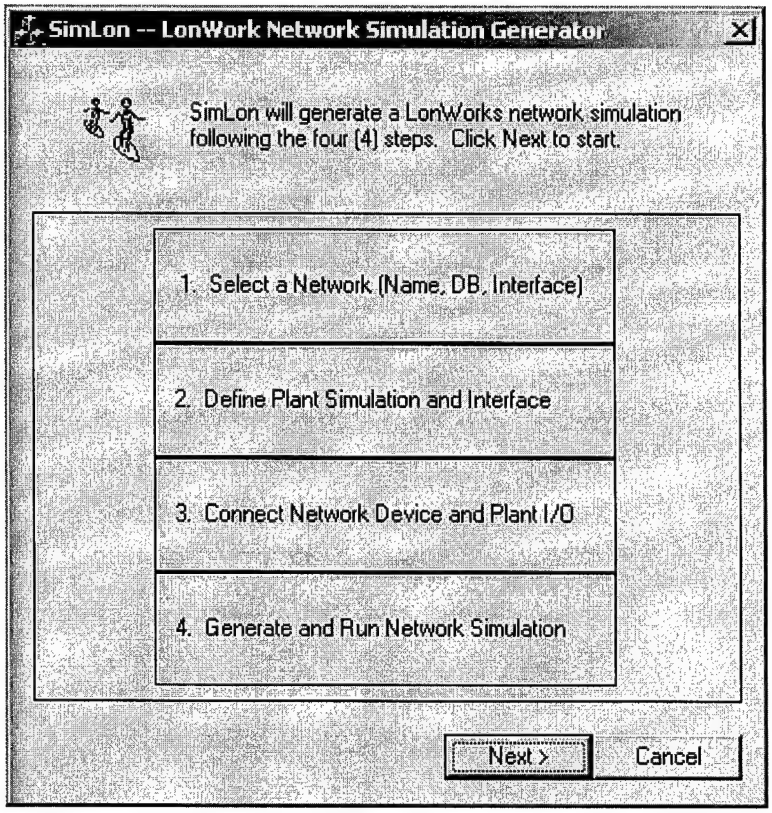


Figure 9 LonWork Simulation Generator Main Dialog

(1) Select the Network.

Click the Next button and the Open Network Database dialog will appear as shown in Figure 10. Select the Resistor_Fan network database that you created and select the correct network interface. Press OK to extract the network database. If a format file is defined, specify the path of the file in the Format File Path. Since no format file is used in this case, this entry is left blank.

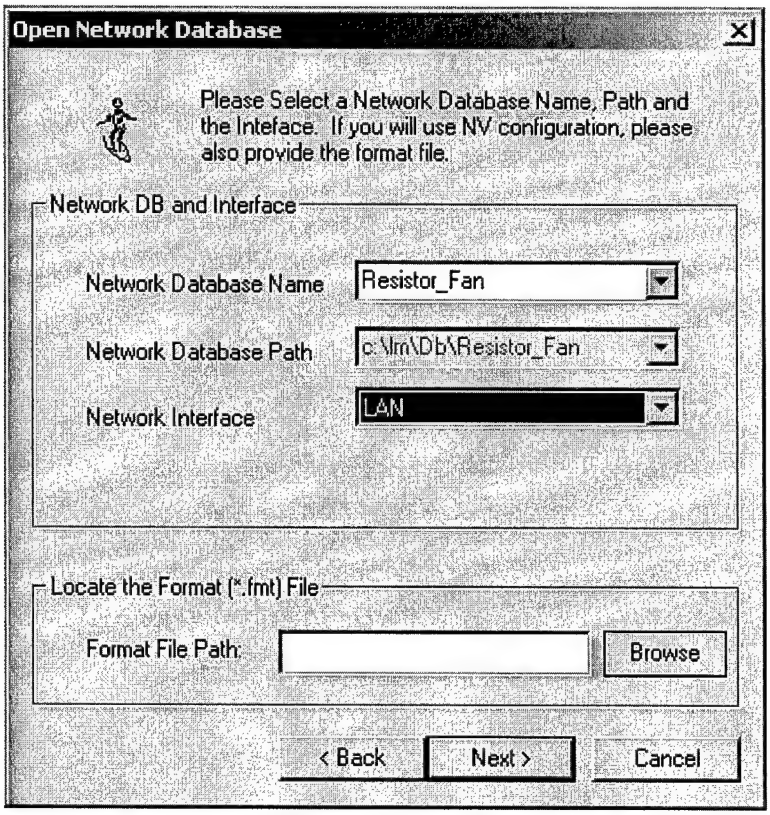


Figure 10 Open Network Database Dialog

The network extraction will search for the device application source file of each device template. If the file is not found, user input will be required. Select the correct Neuron C source files defined in LonBuilder. A progress window will appear to report on processing progress. When this step is complete, click on the *Next* button to define the plant simulation.

(2) Define the Plant Simulation

The plant simulation definition dialog is used to specify the plant simulation code, the simulation update period, and the plant I/O. The plant simulation update time is defined inside the simulation code. Three steps follow to define the plant simulation. Follow the three steps in the Plant Simulation dialog shown in Figure 11 to enter the plant simulation name, select the plant simulation file, and define the plant I/O in the Plant Simulation dialog.

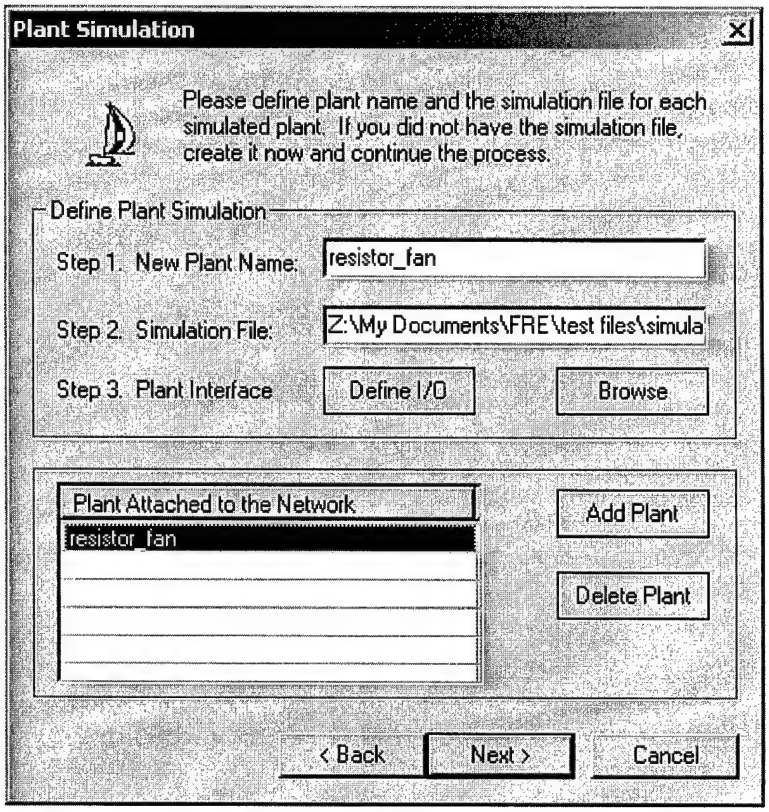


Figure 11 Plant Simulation Dialog

Click the *Define I/O* button in the Plant Simulation dialog. The Plant Interface dialog will open for plant I/O definition as shown in Figure 12. The plant simulation is added to the simulation automatically when opening the Plant Interface dialog. When a plant simulation name is selected, all the variables defined in that plant simulation code will be listed in the Variables list. Select the output or input variables and press the `Apply` button to record the I/O definition. Click `Close` to close the Plant Interface dialog and return to the Plant Simulation dialog.

If the user wants to monitor the value of a variable that is not an I/O signal, select the variable as Output. For example `temp_loop[0]`, `temp_loop[1]`, `temp_loop[2]` in the following dialog are not plant I/O signals but internal variables. These variables are selected in the Output list, which allows the user to monitor their values during the simulation. These variables may not connect to a device I/O point.

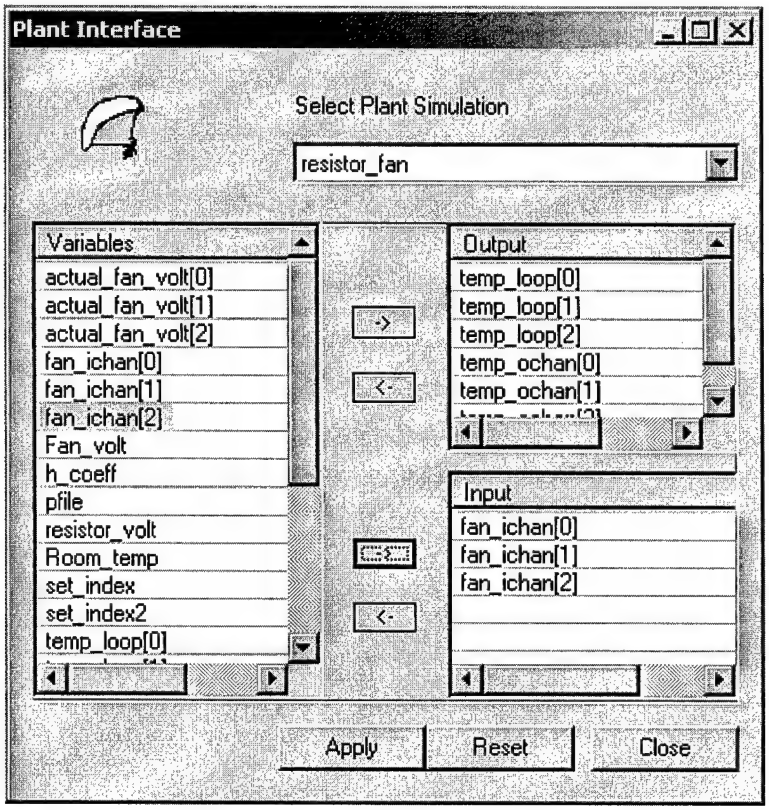


Figure 12 Plant Interface Dialog

After defining the plant simulation, click the `Add Plant` button to add the plant simulation to the list (If you defined the plant I/O signals, the plant simulation is automatically added to the simulation.) Click the `Next` button to go to the Device and Plant Interface Connection dialog.

(3) Device and Plant Interface Connection.

In the Device and Plant Interface Connection dialog, select the device I/O signals and the plant I/O signals which you would like to observe during a simulation. Click the `Connect` button to connect the selected I/Os. All the connections are shown in the Device/Plant Interface Connections list. Press the Delete button to delete a selected connection. Remember that we have defined the `temp_loop[*]` internal variables as plant I/O signals to monitor their values. Connect these variables to unused device I/O points. When finished, click the `Next` button to generate the network simulation definition.

A progress dialog shows the network simulation generation progress (Figure 14). When the process is complete, the next step is to compile the created files and build the network simulation.

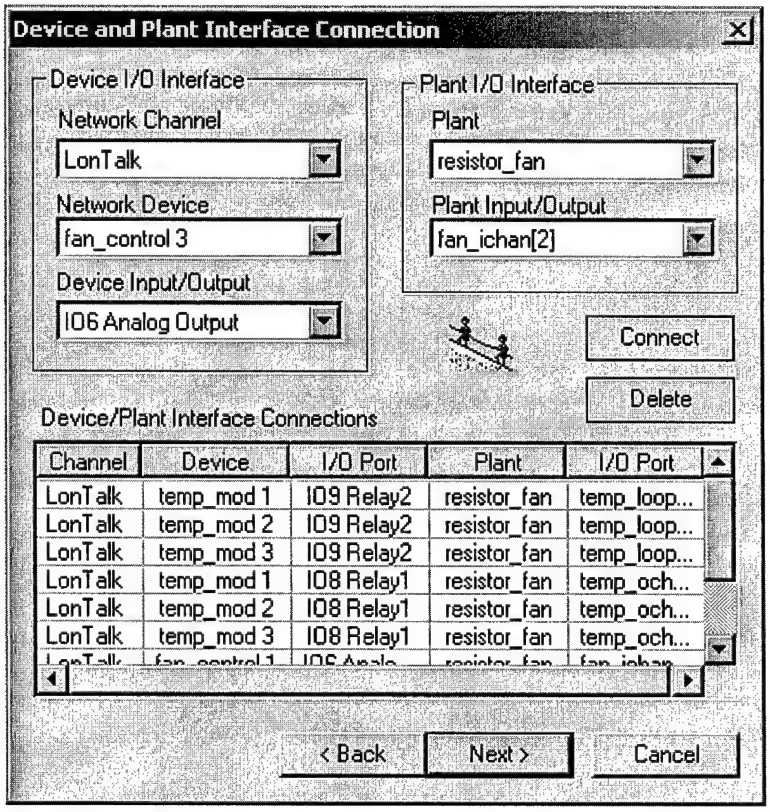


Figure 13 Devices and Plant Interface Connection Dialog

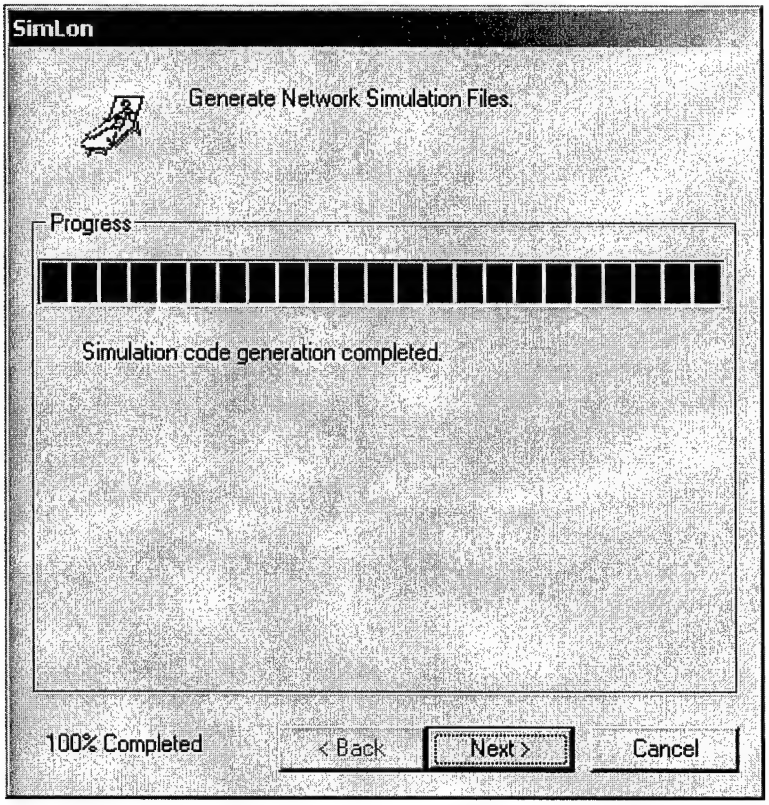


Figure 14 Simulation Generation Progress Dialog

(4) Network Simulation Generation.

Add the generated simulation files to the simulation platform project and open the project in an IDE, such as VC++ as shown Figure 15. Press the build project button in the IDE to compile the project and generate the simulation code for the application. The code is generated in the current working directory. Close the simulation generator. After compilation, open the simulation platform to run the simulation.

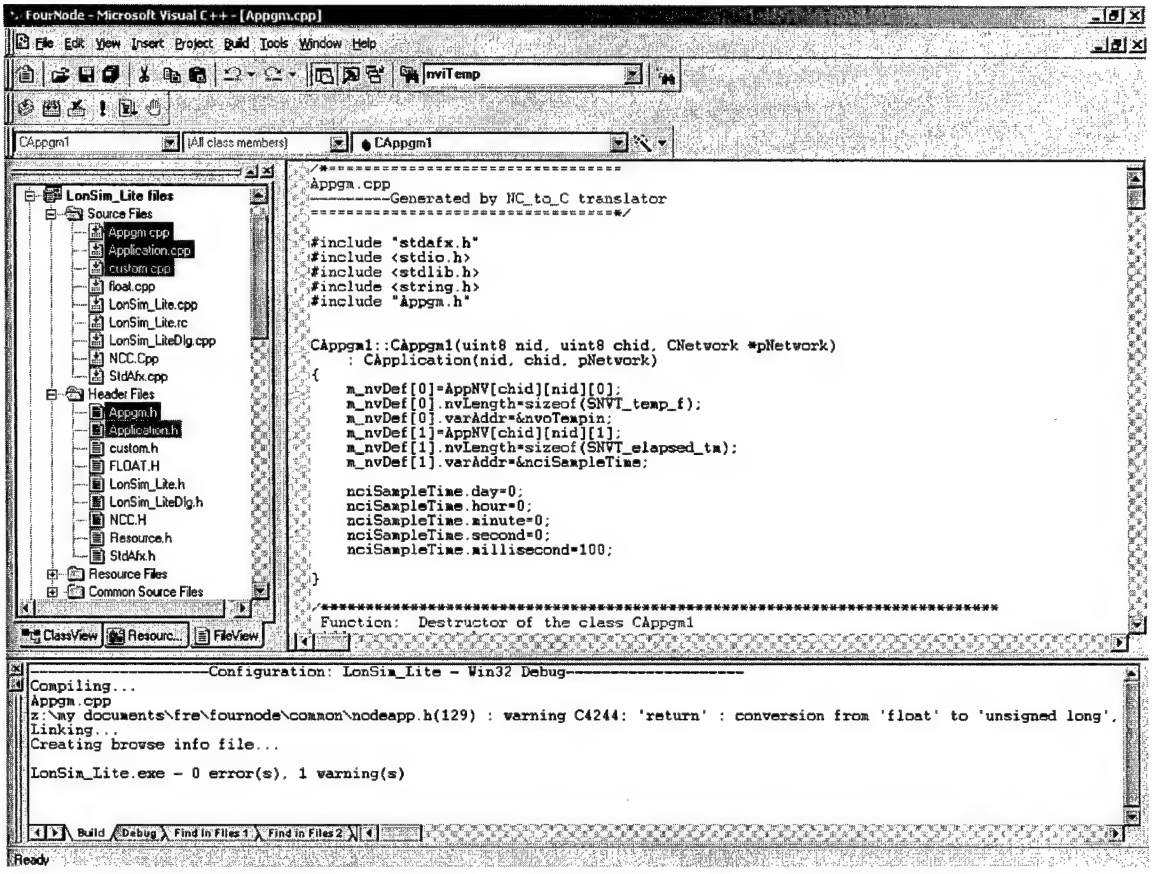


Figure 15 Complication of the Simulation

4. NETWORK SIMULATOR

4.1 *Network Simulator Architecture*

The Network Simulator consists of the VDCS Network Simulation engine and a Graphical User Interface (GUI). Figure 16 shows the architecture of the Network Simulator. The Network Simulator dynamically constructs simulation objects, based on the network simulation definition created by the Simulation Generator. The simulation definition files contain the network configuration and the network application definitions. The GUI module provides a user interface as well as various network analysis tools. The GUI module accesses the VDCS Network Simulation to obtain the simulation status and statistical performance data for display within various analysis tools. The GUI module also allows the user to send commands to the Network Simulation engine, in order to control the simulation process.

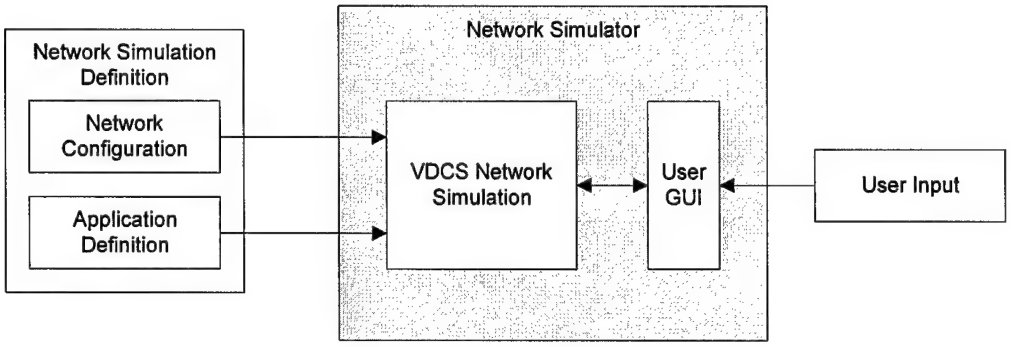


Figure 16 Network Simulator.

The VDCS Network Simulation is divided into several modules implemented in Component Object Model (COM) components. Each module is implemented as a dynamic link library (DLL). These DLLs are loaded when a simulation starts. Interfaces are defined for each module to provide entry points for other modules. Figure 17 shows the definition of the Network Simulation Component or *Simulation Core* that coordinates the network simulation. The plant application simulation and the device application simulation are implemented in individual components. The Network Simulation Component provides *IDeviceAPI* and *IPlantAPI* interfaces for the device application simulation module and the plant application module respectively. It also provides an *ISimulation* interface for the user interface application to invoke and control the Network Simulation component. There are also a number of interfaces that support network simulation operations and network analysis tools.

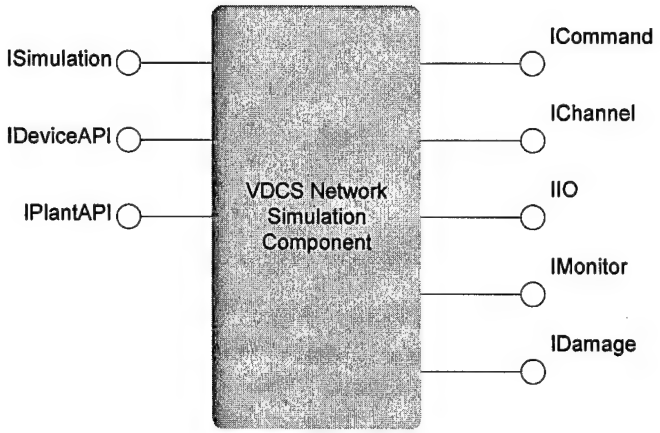


Figure 17 Network Simulation Component

A detailed architecture of the Network Simulator in terms of COM components is shown in Figure 18. The Simulation Host is the user interface, and the GUI also consists of network analysis tools including Network Analyzer, Application Monitor, and Damage Control.

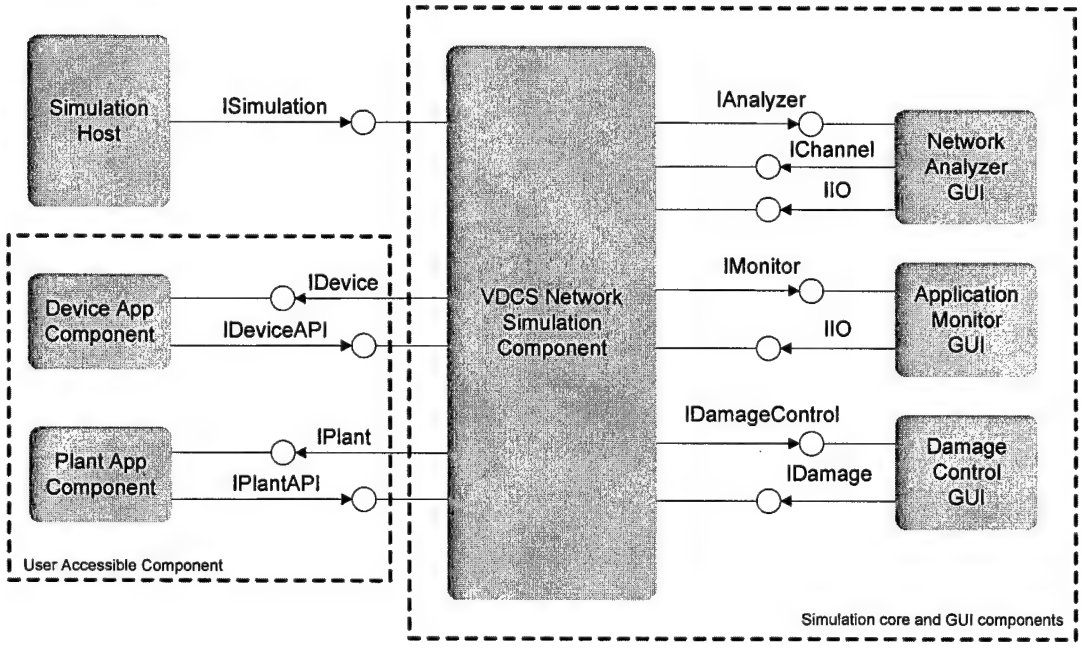


Figure 18 Network Simulation Architecture

4.2 Channel Model

The channel model represents the physical channel in a network. The channel is assumed to be a continuous cable or wire that connects network devices in the form of a chain or a ring.

The maximum length of a channel specified by LONWORKS limits propagation delays and mitigates their effect on communication errors. We have decided not to include the effect of propagation delays on our channel model. Since we are interested in the effects of cabling damage on the network (e.g., a connection between devices is severed), we had to incorporate the network topology into our model as well as a mechanism to describe message propagation.

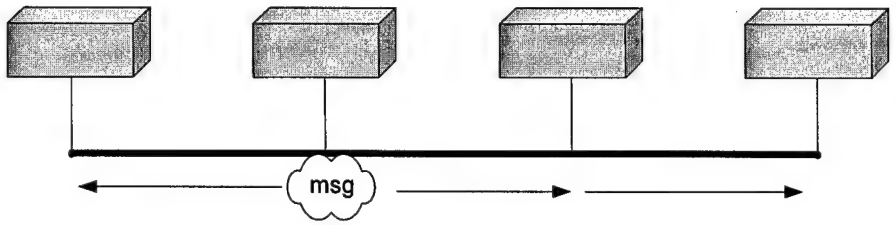


Figure 19 Message Propagation along a Channe

Our channel model consists of buffers for each device, attached to the channel. Each buffer maintains a local copy of the last packet propagated through the channel. As shown in Figure 19, when a device is ready to transmit a packet, it copies the packet into its channel buffer and propagates it to adjacent buffers. The packet is transmitted to the device next to the sending device and then propagates to other devices down the channel. This mechanism is repeated in the channel model by propagating the whole packet not "one bit at a time". The channel state is also propagated from buffer to buffer.

The procedure of packet propagation is demonstrated in Figure 20. The figure shows four devices connected to a channel. The double ring represents the service that the channel model provides to the devices. The channel model allocates a buffer for each attached device. The packet transmission has four phases as illustrated in Figure 21. First, a packet is generated in Device 1 and transmitted to the channel. Then the channel makes a copy of the packet in the buffer assigned to Device 1. Next, the channel propagates the packet to the buffer of each device according to their connection order. The signals propagate across the channel in both clockwise and counter-clockwise directions. A separating point is defined to prevent a message from looping back to the source. The separating point is set between the first device and the last device of a channel. When the propagation in each of the two directions reaches the separation point, it is terminated. Finally, each device on the channel receives the packet from its local channel buffer. This mechanism guarantees that the packet is propagated to every channel buffer only once.

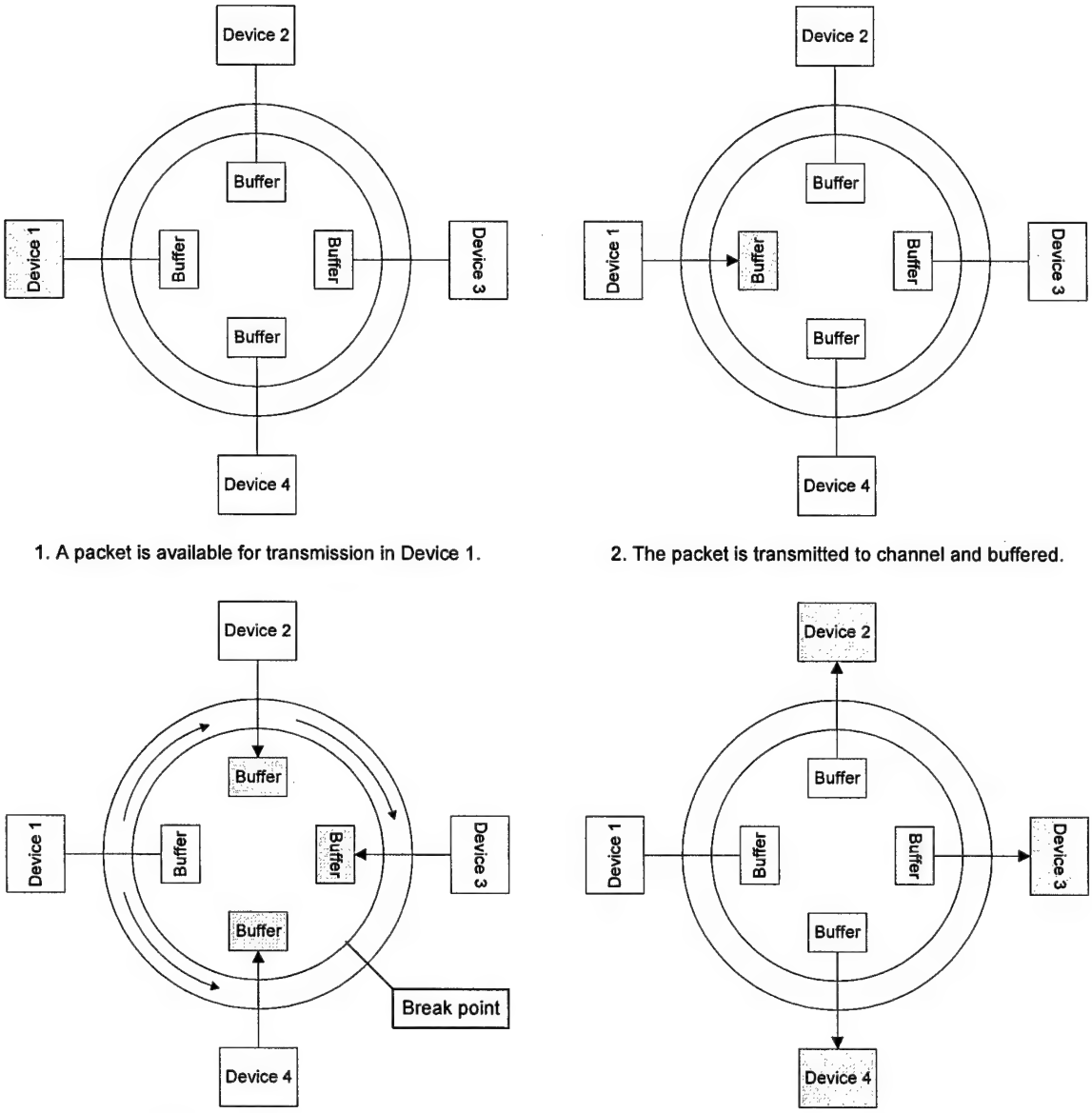


Figure 20 Packet Propagation Through a Channel

4.3 Channel Types

All of the LonTalk channels use the ANSI/EIA 709.1 [9] communication protocol standard. Different types of LonTalk channels employ various transceivers with different transmission rates and timing parameters. Echelon Inc. provides a software application called PERF to evaluate the transceiver timing parameters. The timing parameters that characterize the channel performance are Beta 1 time, Beta 2 time, priority slot number, bit rate and clock rate. To model the different channel types, we set up a look-up table that contains all the timing parameter values for each

channel type. When initializing a network simulation, the channel timing parameters are set according to the channel type. In the present prototype, we have implemented two channel types: TP/FT-10 and TP/XF-1250. We have extensively tested the TP/XF-10 channel in both Phase I and Phase II of VDCS development.

4.4 Ethernet Model

Ethernet is a popular communication protocol for high-speed data transmission. The EIA 852 standard [1] defines the protocol for LonTalk over Ethernet. Ethernet gateways, such as the iLon Gateway from Echelon, are used to connect a LonTalk channel to an Ethernet network. Since data transmission rates over Ethernet channels are significantly higher than they are over TP/XF-10 LonTalk channels we have assumed a simplified model for the Ethernet channel.

The Ethernet simulation module is based on three assumptions: (1) the transmission rate of Ethernet is significantly faster than that of LonTalk channel, so that the transmission time of Ethernet can be effectively ignored; (2) the Ethernet channel is merely used as a communication backbone without any application devices attached to it; (3) there are small number of LonWork channels connected to the Ethernet channel, so that the traffic load on the Ethernet channel always remains low.

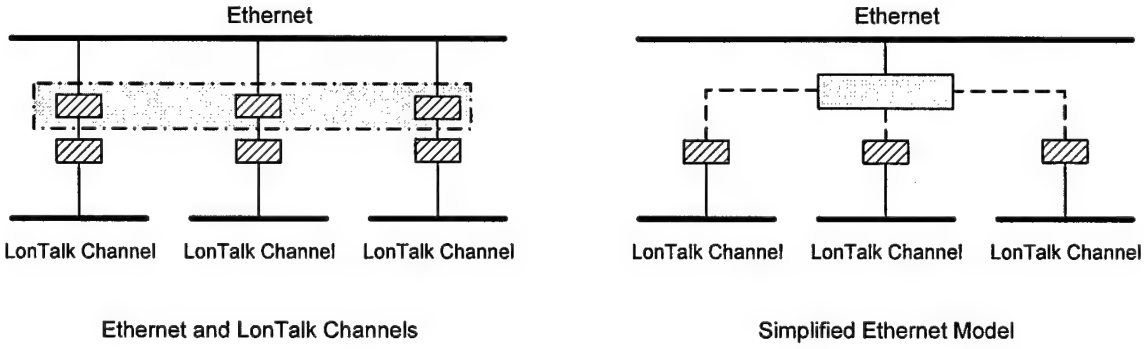


Figure 21 Ethernet and LonTalk Channels

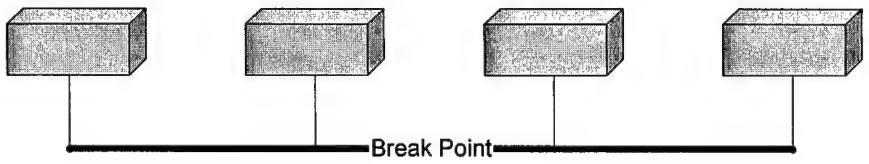
The left-hand side of the drawing in Figure 21 shows three LonTalk channels connected to an Ethernet channel via three routers. A router has an "A side" and a "B side" that connect the two different channels. Section 4.6.2 describes how we implemented both sides of a router that connects two LonTalk channels with the same protocol stacks used for a LonTalk device. For a router that connects an Ethernet channel to a LonTalk channel, we merged all the router stacks that connect to the Ethernet channel into a single object. We also incorporated the single router object into the Ethernet model. As shown on the right-hand side of Figure 21, the router object connects all of the router stacks that connect to the LonTalk channels. When a message is sent from one LonTalk channel to another LonTalk channel via the Ethernet link, the message is initially sent to the router connected to the source LonTalk channel. The message is then stored in the router stack and is propagated to the router object of the Ethernet model. The router object will check the addressing of the message and pass it along to the router that is connected to the destination LonTalk channel.

Our Ethernet model assumes that there is no delay in message transmissions. The VDCS simulation engine therefore does not increment time as a message propagates along the Ethernet channel. The LonTalk router model that forwards the message to the Ethernet channel is used to drive the Ethernet simulation. As a result, the Ethernet model, unlike the LonTalk channel model, is not synchronized by the network synchronization algorithm. As a result the introduction of Ethernet module did not affect the original synchronization algorithms. The network thread synchronization is explained in section 4.7.

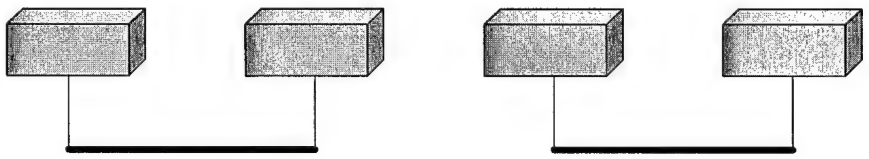
4.5 Network Damage

One of the objectives of the VDCS platform is to simulate network damage events. We consider three types of network damage scenarios: (i) channel damage, (ii) router damage, and (iii) device damage. Channel damage could result from the severing or short-circuiting of device-to-device wiring; when it occurs, it divides a single channel into two distinct channels. A message cannot pass through the break point. Router damage occurs if the router device malfunctions or if there is a broken connection between the channel and the router. A message cannot pass from one channel to another through the damaged router. The device damage scenario represents a disabled device either because of power loss or malfunction.

Damage to routers or devices is relatively easy to incorporate into the router and device simulation models—they simply stop responding. In this section, we focus on the implementation of the channel damage scenario. As shown in Figure 22, damage to the channel divides it into two distinct channels. Functionally, the messages on one side of the original channel are not able to reach the other side.



Before Channel Damage



After Channel Damage

Figure 22 Channel Damage Scenario

As described in section 4.2, the message and channel state for each device are held in an associated channel buffer upon transmission. The message is not made available globally for every buffer on the channel. Instead, the message and the channel state are propagated to adjacent buffers one after another according to their physical network layout. Message propagation stops at a break in the channel.

To introduce channel is halted during a simulation, VDCS marks the message buffer(s) closest to the severed link as damaged. Message propagation will terminate at buffers marked "damaged". The buffer marked as "damage" will not get the copy of the message and will not propagate a message to its neighboring buffers.

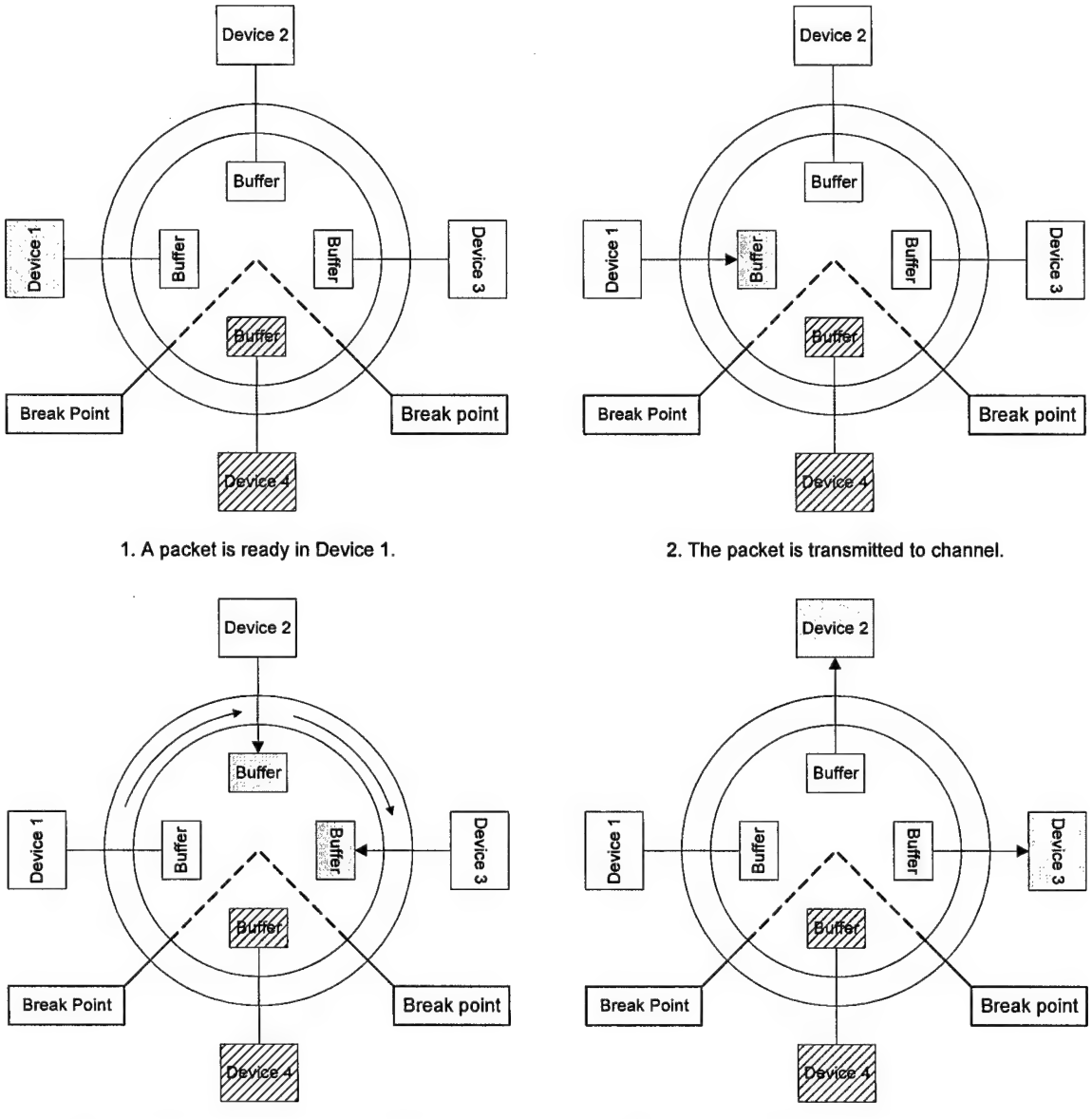


Figure 23 Packet propagation in a damaged channel

Figure 23 illustrates message propagation along a damaged channel. In the figure, a channel discontinuity occurs between Device 3 and Device 4. Message packets are propagated in both directions along a channel. A separating point is used to define the two ends of a channel—in this case, between Device 1 and Device 4. The separating point and the damage point form the two breakpoints that isolate Device 4, preventing it from receiving the packet from Device 1.

4.6 Multi-Channel and Router

When a network has more than one channel, devices in different channels can only communicate with one another via routers or an iLon server. A router connects two channels and forwards messages back and forth according to a routing algorithm. In this section, we describe the router model and Ethernet router implementation used in VDCS

4.6.1 *Router Model*

VDCS uses a hierarchical form of network representation similar to the LNS Database. Devices and routers are arranged as branches of a channel. Since a router connects two channels, the LNS database may associate two channel objects with the same router that connects them. However, VDCS simulates physical interactions between a channel and a router. It cannot allow a router module to be duplicated. To design the channel simulation as an independent entity, it must separate the connection of a channel to other channels connected by a router. We also need to define the routers connected to a channel in the channel configuration, as well as which channel the channel is connected to through the router.

We analyze here the router structure in order to gain insight about partitioning of a routed network. A router has two neuron chips which function as independent devices exchanging messages with each other and the channels they attach to, as shown in Figure 24. Each neuron chip has its own protocol stack—called the Router Stack.

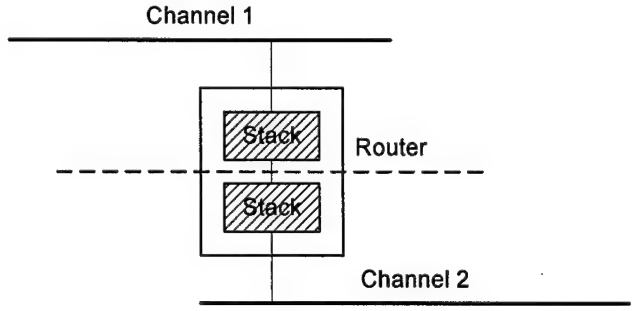


Figure 24 Router Structure

The design facilitates the separation of one channel simulation from another. Each router stack is directly connected to a channel, so that it can be grouped with the other devices on the channel object. The dashed line in the middle of Figure 24 represents a boundary between the two channels.

Figure 25 shows a more detailed diagram of a communication router that links two sub-networks. The router implementation contains two sets of the same bottom three layers found in a standard processing node implementation. Each layer set is connected to one of the two channels that the router links. Similar to the device model, the router stack contains a network layer and a link layer, as well as a MAC sub-layer that interfaces to a physical layer. The network layer determines if a received message should be forwarded to the other channel according to a routing table. If no forwarding is necessary, the message is dropped. The routing algorithm is based on the EIA 709.1 network protocol. The implementation of the algorithm is described in the next section.

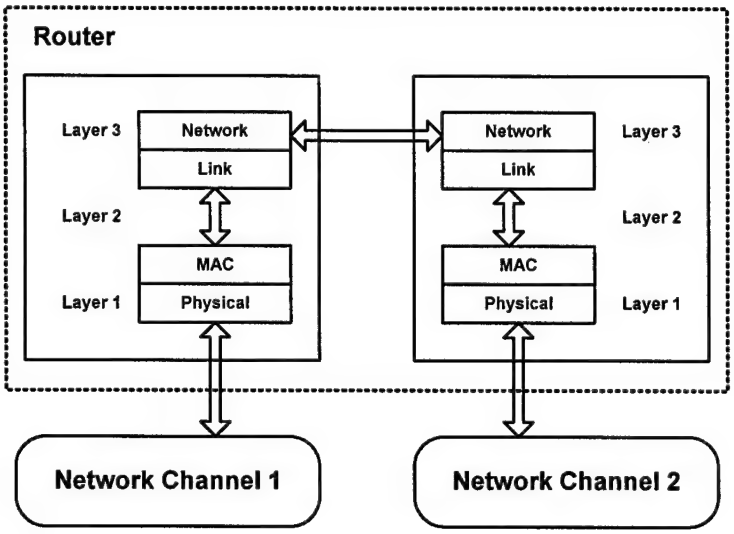


Figure 25 Software Architecture of Router

VDCS assigns a message buffers to each router stack. The two stacks in a router communicate by accessing each other's buffer. The channel configuration contains a router list that defines the routers connected to a channel. When a channel is initialized, the router list of each channel is scanned. If two channels contain the same router, a connection is made by exchanging their message buffer addresses.

A router can be configured as one of the following types: (1) configured router, (2) learning router, (3) bridge, (4) repeater, (5) permanent bridge, and (6) permanent repeater. The permanent bridge behaves same as a bridge, except that its type cannot change after creation. Similarly, the permanent repeater is functionally equal to a repeater but the type is fixed. Therefore there are typically four types of routers. The router model can be configured as the specified type in a network design. The type of router is set during network initialization and can be changed using a function if necessary.

4.6.2 *Routing Algorithm and Forwarding Tables*

A *repeater* transmits every received message from one channel to another. A bridge forwards only those messages destined to a fixed domain. A *configured router* and a *learning router* forward or discard messages according to a routing algorithm. The routing algorithm as defined by the EIA 709.1 standard [11] is illustrated in Figure 26.

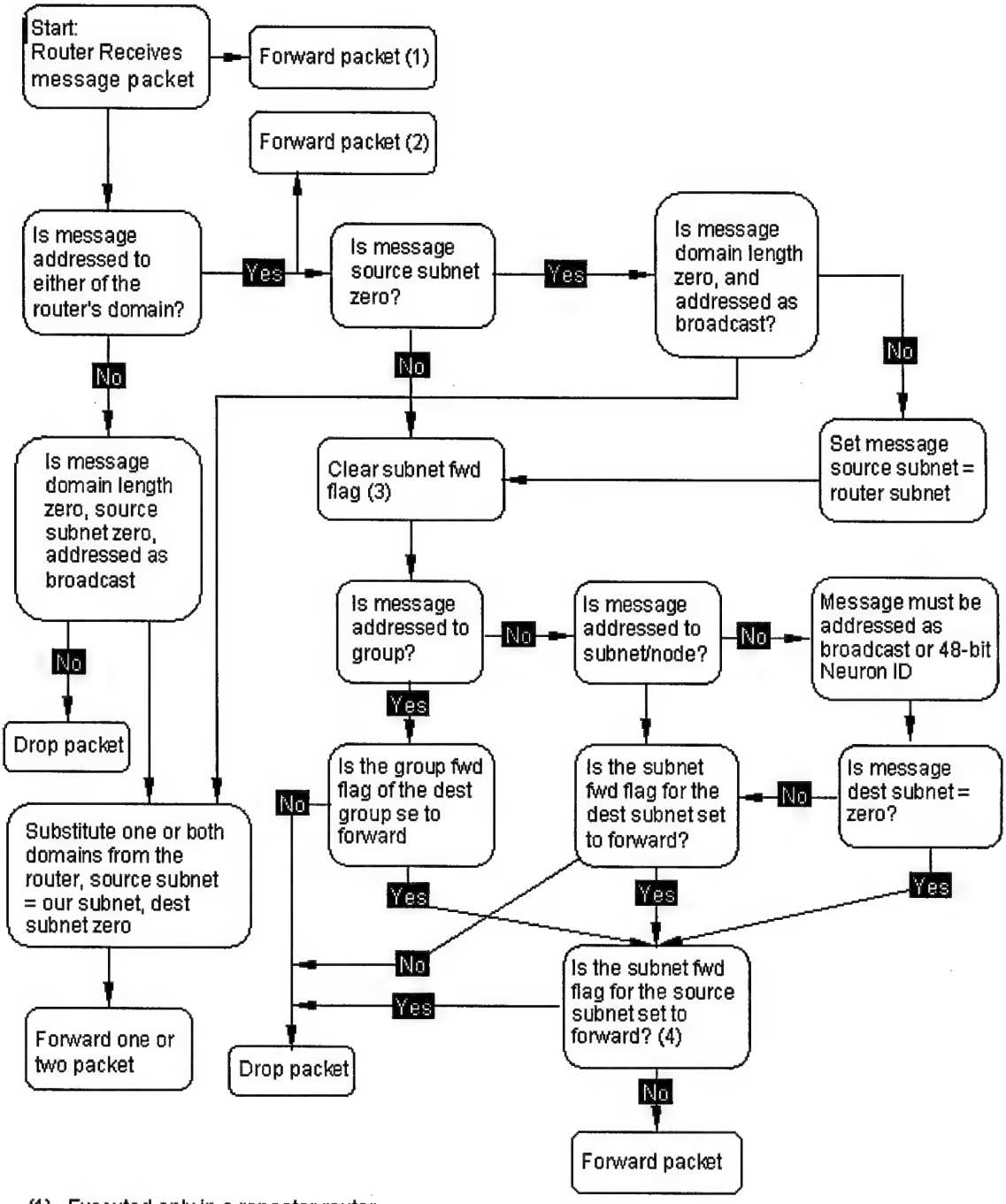
The routing algorithm contains two built-in forwarding tables used to make forwarding decisions. The two tables are the subnet forwarding table and the group forwarding table. The subnet forwarding table contains the subnet information and defines the set of subnets to which a message destination will be forwarded. The group forwarding table contains the set of groups to which a message destination associated with that group will be forwarded.

We defined the two forwarding tables in the router stacks. Each router stack keeps a set of forwarding tables individually. Each entry of these tables represents a network subnet or a group. A flag is used in each entry to indicate if a message sent to the subnet or group should be forwarded or discarded. Only the messages to those subnets or groups that have the set flag will be forwarded.

The router stack forwarding tables of a configured router will be set by the Simulation Generator according to the network topology. A search algorithm is used to explore the network hierarchy to find all the subnets and groups of messages in the part of a network where the router stack forwards messages. The search algorithm is implemented as a tree search. The search starts from the router that the stack belongs to and goes to the other side of the network that the router stack does not directly connect to. Figure 29. illustrates a search that starts from the shaded router stack and searches the network in the highlighted area. If a router is identified, the channel that it is connected to is recorded. When the channel is recorded, the router is flagged as processed. Then the new channel is searched to discover additional routers that have not been marked as processed. When the channel search is complete, the channel is marked as processed. Discovered routers generally yield additional channels to search. The search terminates when each and every channel and router is marked as processed.

During the search, subnets are checked by examining a router stack's subnet ID. Groups are checked by going through each entry of every device address table. A flag is set based on the existence of a subnet and a group in the destination side of a router stack in the corresponding forwarding table.

The subnet forwarding table of a learning router is set according to the transmitted messages during the network simulation. The router will learn the subnet location based on the address of sending devices. The group forwarding table of learning router is not configurable, thus any group message will be forwarded.



(1) Executed only in a repeater router.
(2) Executed only in a bridge router.
(3) Executed only in a learning router.
(4) Executed in a configured router, otherwise forward

Figure 26 Routing Algorithm

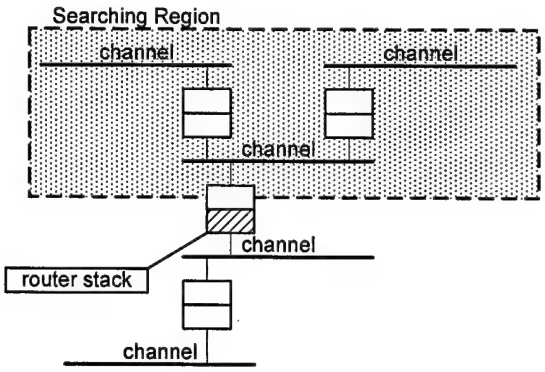


Figure 27 Searching Through a Network to Set the Forwarding Tables

4.6.3 *Router for Ethernet*

An Ethernet router or gateway is divided into two parts: a LonTalk router stack and an Ethernet router stack. The implementation of the LonTalk router stack was described in section 4.4. Since we implemented a simplified Ethernet model, all of the Ethernet router stacks of an Ethernet channel are simplified and are represented as a single router object. The router object has one *In Stack* and one *Out Stack*, as shown in Figure 28. The In Stack is connected to every LonTalk router stack by sharing its buffer memory, so that every LonTalk router stack may forward a message directly to the In Stack. The Out Stack connects to a LonTalk router stack by holding the buffer address of that router stack.

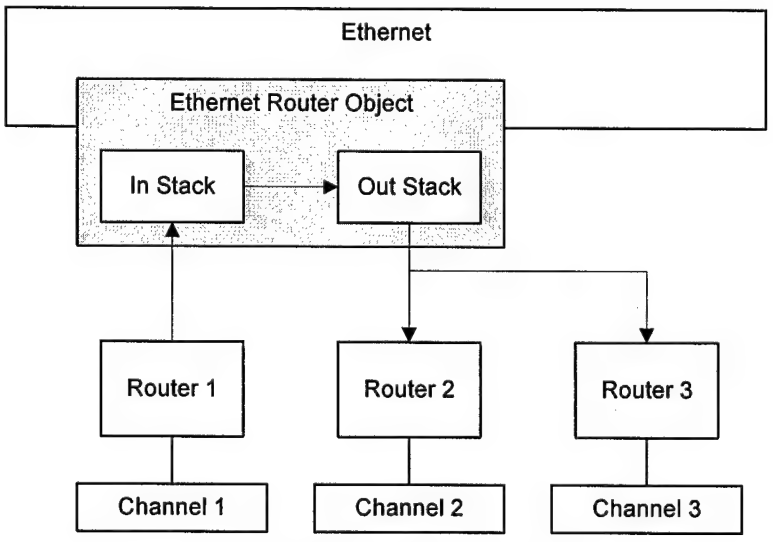


Figure 28 Ethernet Router Object

When a message is forwarded (as illustrated in Figure 28), the LonTalk router stack (Router 1) sends the message to the In Stack of the Ethernet router object. Then the Ethernet Router object forwards the message to the Out Stack. Then, the Out Stack dynamically connects to each of the LonTalk routers and determines if the message should be forwarded to that LonTalk router, one at a time. First the message is copied to Out Stack; then the Out Stack connect to one of the LonTalk router stacks; finally, the Out Stack determines whether to forward the message to that router stack. The message will be copied to the Out Stack again for forwarding the message to another router stack until attempts are made to forward the message to all the router stacks. If the router stack is the one sending the message, the Out Stack will skip it and continue to check the next router stack.

4.7 Simulation Threads, Time and Synchronization

The VDCS test platform is designed as an event-driven multi-threaded simulation software package. During a simulation run, each thread performs a subset of the computation effort synchronizing the time evolution of each thread is critical to maintaining causality. During Phase II of the VDCS development effort, several changes were made to the thread synchronization algorithm to accommodate the expanded simulation features (e.g., incorporation of routers, redesign with COM technology, and improved channel model.)

4.7.1 *Simulation Threads*

A typical network consists of several channels, devices and routers. A single channel and all of the devices and routers that are connected to that channel together form a *subnet*. A collection of subnets forms a *network*.

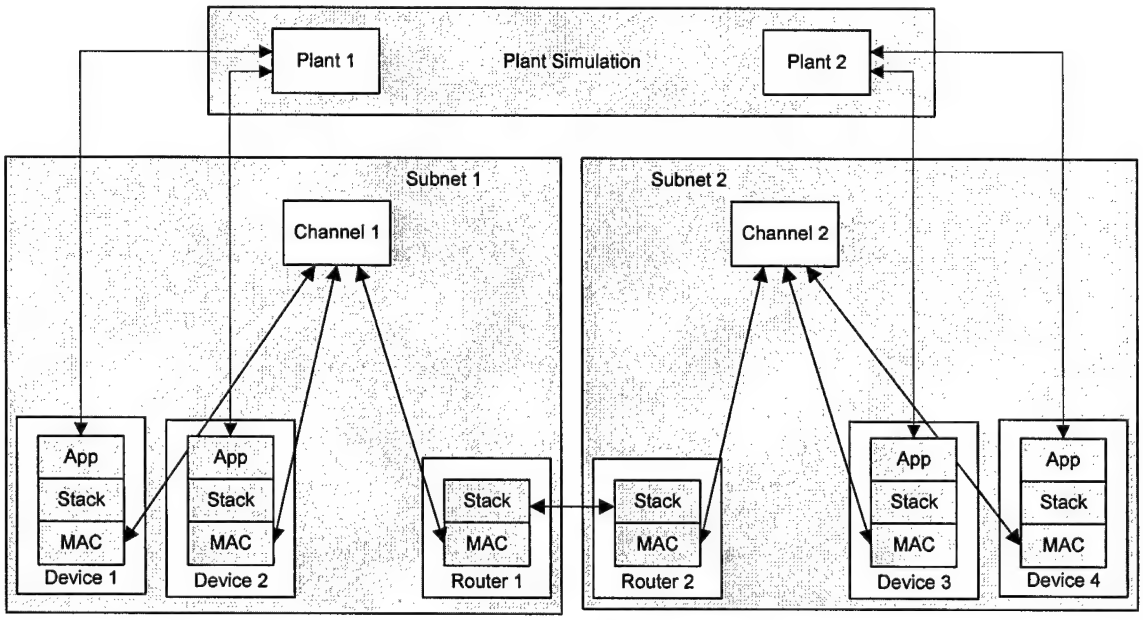


Figure 29 Network Simulation Threads

VDCS uses a network object and various threads to manage the network simulation. One thread is used to simulate each channel. Three threads are used to simulate each device, one for the device application, one for the device communication stack, and one for the device MAC layer. Two threads are used to simulate each router, one for the router stack, and the other for the router MAC layer. All of these threads simulate the network internal events in the order of time that they occur. A network simulation controller controls most of the threads, except for the MAC layer threads. The channel thread synchronizes all of the MAC layer threads that access that channel. The plant simulation is also represented as a thread, which is synchronized with the network simulation. Figure 29 represents a network simulation in terms of threads. Each block represents a simulation thread for a simulation object. Data are exchanged between threads that are next to each other in the figure. The lines connecting the functional blocks represent data flow (e.g., message transmissions) between the various network components.

4.7.2 *Simulation Time Update*

Each network simulation thread maintains a local simulation clock to keep track of their simulation progress. A network simulation controller runs the synchronization algorithm to control the simulation threads according to their simulation time. The simulation time of a thread is called the local simulation time. The simulation controller also maintains a global simulation clock to keep track of the overall simulation progress. The global simulation clock evolves with the progress of any local simulation clock. Every time an event is simulated in a thread, its simulation time is updated. The simulation controller only updates the thread that results in the smallest time increment of the global network simulation clock. The global network simulation clock evolves according to the local simulation clock of the thread that is currently updated.

The simulation clock is implemented as a counter with fixed size (i.e., the counter has a maximum number that it can represent). With the fine granularity used in VDCS simulations, the current maximum time that the clock can represent is about 12 hours. When the clock value exceeds the maximum value, it overflows. This overflow event is handled with a dedicated procedure. The global simulation clock overflow event begins when a local thread simulation clock overflows. When all of the thread simulation clocks overflow, the network simulation clock overflows. When a thread clock overflows, an overflow flag is set in the thread. The status of the flag is reported to the network simulation controller. The simulation controller converts the overflowed clock value to its actual value and makes a decision to update the threads. The simulation controller keeps synchronizing the threads until all the local thread simulation clocks overflow. When all the thread simulation clocks overflow, the simulation controller clears the overflow flag of each thread, terminating the clock overflow event.

4.7.3 *Thread Synchronization Scheme*

While the synchronization algorithm developed in Phase I successfully coordinated the simulation threads in VDCS for a single channel network, it had some shortcomings. The algorithm did not scale well with increasing numbers of channels and devices in a network. In addition, the algorithm did not handle efficiently simulations characterized by threads with significantly different local time increments. This happens frequently in VDCS simulations where the channel threads evolve in fine time increments while the device application threads evolve at relatively much larger time steps. A new synchronization algorithm was developed in phase II to address these shortcomings.

The new synchronization algorithm uses a scheme to predict the local time increments in order to improve performance.

The new synchronization algorithm allows a thread to update only if it makes the minimal time increment. The simulation task is divided into a set of events. Each thread simulates one event when it updates. Each thread increases its simulation time by the amount that the event costs. A simulation controller governs the synchronization activities. Every time a thread attempts to update, it calls the simulation controller and reports its current time and the time increment that would result from executing the next event. An overflow flag is also passed to the controller in case a clock overflow event occurs. The simulation controller checks every thread during a thread update cycle, and maintains a list of all the threads' next time increment. Only the thread(s) with the minimum time increment are permitted to update during a thread update cycle.

A thread may execute repeatedly while other threads are idle. The previous algorithm checked every thread at every time increment before allowing a thread to update. The new algorithm provides each thread with self-awareness mechanism to determine if its future updates still represent the minimum time increment among all simulation threads. This mechanism allows a thread to execute several events in succession without reporting the result to the simulation controller.

Communication between threads is another critical issue in thread synchronization. Thread communication occurs between: (1) the device application and the plant application, (2) the device application and the device stack, and (3) the device/router stack and the channel (the device stack communicates with channel through the device MAC layer). VDCS follows two rules to guarantee that the simulation maintains causality.

> *Rule I*: For any Logic Process (LP), at any time when it receives messages from other LPs, the current time of this LP must be larger than the timestamp of any available messages.
>
> *Rule II*: For any LP, at any time when it outputs a message, the timestamp of the message must be larger than any other available messages.

The synchronization algorithm must guarantee that a thread receives the latest information from other threads. No future information should be received. Therefore the synchronization needs to strictly control the communication between threads. The improved algorithm requires a thread to receive information and update its output before simulating a new event, since completing the event will increase the local simulation time.

We use here the same nomenclature introduced in the final Phase I report that described the synchronization algorithms. We define k_i as the k^{th} execution of the i^{th} thread, $T_i \quad (i = 1,\ldots n)$. $T_i(k_i)$ represents the simulation task at time $t_i(k_i)$. We further define $t_i(k_i)$ and $\Delta t_i(k_i)$ as the simulation time and time increment respectively, of the i^{th} thread T_i at its execution step k_i. We specify that t_0 represents the global simulation time to which all other $t_i \quad (i = 1,2,\ldots n)$ parameters are synchronized.

Table 3 describes the synchronization algorithm executed by the network simulation controller.

Table 3 Simulation Controller Synchronization Algorithm

STEP 1. Initialize $t_0(0)=0$, $\Delta t_0(0)=0$, $k_0=0$.
STEP 2. Check all thread T_i for $i = 1, 2, \ldots, n$.
For each i satisfying $t_0(k_0)+\Delta t_0(k_0)=t_i(k_i)+\Delta t_i(k_i)$:
A) Enable the thread simulation.
B) Record $t_i(k_i+1)+\Delta t_i(k_i+1)$
STEP 3. Set $t_0(k_0+1)=t_0(k_0)+\Delta t_0(k_0)$
STEP 4. Set $\Delta t_0(k_0+1)=\min[t_i(k_i)+\Delta t_i(k_i)-t_0(k_0)]$ $i=1,2,\ldots,n$
STEP 5. Increment execution step $(k_0=k_0+1)$
STEP 6. Go to STEP 2.

Each thread executes a local algorithm as shown in Table 4 upon being enabled by the simulation controller. The algorithm employs a look-ahead technique. The initial value of $\Delta t_i(k_i)$ is set to 0, to enable all the threads to execute when the simulation starts. The current $\Delta t_i(k_i)$ is determined based on the execution of a simulation task $T_i(k_i+1)$. Based on $t_i(k_i)$ and $\Delta t_i(k_i)$, the simulation controller can determine the next step of the thread T_i.

Table 4 Thread Synchronization Algorithm

STEP 1. Initialize $t_i(0)=0$, $\Delta t_i(0)=0$, $k_i=0$, stop = true.
STEP 2. Wait for being enabled for simulation.
STEP 3. Enabled for execution: Do A), B), and C) in order.
A) if stop = false and $t_0(k_0)+\Delta t_0(k_0)>t_i(k_i)+\Delta t_i(k_i)$,
stop = true, and go to STEP 2. Otherwise,
B) if $t_0(k_0)+\Delta t_0(k_0)\neq t_i(k_i)+\Delta t_i(k_i)$, go to STEP 2. or
C) (1) $t_i=t_i(k_i+1)=t_i(k_i)+\Delta t_i(k_i)$,
(2) Process simulation $T_i(k_i+1)$,
(3) Determine $\Delta t_i=\Delta t_i(k_i+1)$, and
(4) stop = false.
Repeat STEP 3, until exit to STEP 2.

A verification test was developed for the synchronization algorithm to evaluate the effectiveness of the improvement. The verification test compared the overhead of the simulation synchronization algorithm added to the network simulation. The test is arranged in two scenarios where the thread time increment is different. The test shows that the randomness of the thread's time increments has a significant impact on the simulation speed. A test of four threads synchronized with the improved algorithm showed a 40% performance improvement over the previous version when threads updates with totally random time increments; and a 70% performance improvement when the time increments of the different threads varied by at least 10 times difference between each other. These tests also validated that causality is never violated.

4.8 Network Device Application

The network device application is the application that runs on the network devices to determine the device behavior. LONWORKS device applications are programmed in Neuron C and downloaded to each device after compilation. The device application simulation includes simulating the device application and simulating the way a Neuron Chip executes the application, as well as other related behaviors.

4.8.1 *Device Simulation Implementation*

The implementation of a device application consists of three components: a device application simulation, a protocol stack simulation and a MAC layer simulation. The device application simulation executes a user-defined program that interfaces with local I/O, prepares messages to send over the network, and processes and responds to incoming messages. The protocol stack simulates the EIA709 communication protocol on a LonWork device. The MAC layer runs the media access algorithm to interface to the communication channel.

The device application simulation is divided into a device application service and a device application simulation. Figure 30 illustrates the architecture of the implementation of the device application. The device application service simulates the application services defined by the communication network protocol supported by the Neuron Chip. It works as a middle layer between the device application and the device stack. This part is implemented inside the Simulation Core. However, the actual device application executes on a separate component implemented as a stand-alone DLL called DeviceApp.dll. The device application simulation component simulates the device applications translated from Neuron C code using the API services provided by the device application service. Interfaces IDeviceAPI and IDevice are employed by the device application service and the device application simulation respectively to communicate to each other.

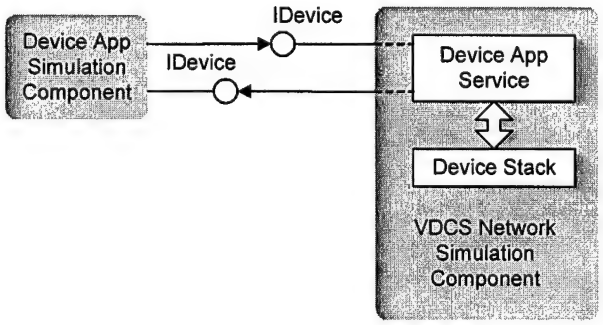


Figure 30 Device Application Implementation

The major motivation to separate the device application simulation from the simulation core is to make the device application debugging easier and to make the network simulation transparent to the software user when debugging the network application code using VDCS.

4.8.2 *Device Application Service*

The device application service executes within the device application thread in the simulation core. It is synchronized with other simulation threads by the network simulation controller. Generally, there are five services that the application service manages as listed below.

- Device application simulation initialization
- Device application simulation execution
- Network variable propagation
- Timer update and timer expiration event handling
- Device IO updates.

It also supports the functions of the device application API for application code execution. The device application API is available through the interface IDeviceAPI of the simulation core for the application service. The API is described in Device Application Simulation API.

The device application service also interfaces with the device stack. It is connected to the application layer and forwards/receives network messages to/from the application layer of the device stack. An event queue is maintained by the device application service. When a message is received, a related event is added to the event queue (if the device application responds to that event). The implementation of the queue is based on tests of *smartcontrol* programmable device.

4.8.3 *Device Application Simulation*

The device application simulation communicates with the application service using an interface Device defined in DeviceApp.dll. The device application simulation implements a simulation of the scheduler used in the Neuron Chip to control the application execution. It is based on the device application source code implemented by the user. The VDCS translates and coverts this source code into a C++ format and compiles it into the device application component. Since the network device application is defined in Neuron C code, the translation task and the code complexity and flexibility requires VDCS to build a device application simulation component for each network simulation.

The simulation generation process is automated with the introduction of a Simulation Generator. The definition of the device application simulation is described in detail in Appendix I. A compiler and a linker are required to build the device application simulation code into the device application component. The VDCS simulation generator uses the console *msdev.exe* compiler included with the Microsoft Visual C++ development environment. It defines a project file for the device application simulation project. When the simulation generator builds a network application, it launches msdev.exe, which loads the pre-defined project file. The device application simulation is built as a DLL according to the settings of the project file.

4.8.4 *Response Time*

The device response time is the speed that the scheduler responds to an event or to propagate scheduled messages. It is represented in the simulation by the device simulation time increment. This time varies according to particulars of the user application code. This device behavior requires a dynamic determination of the simulation time increment and updating the simulation time based on the device application activity. However, this value is difficult to obtain as a number of conditions affect the device response time, including the application code computing time, the number of network variables, and the number of timers in the user application code. The thread synchronization algorithm described in section 4.7.3 supports adaptive simulation time increments in each thread. Although we do not have an effective method to determine the device response time at this time, we developed API functions for the applications to dynamically update the simulation time increment.

4.9 Control Plant Application

The plant simulation model is developed in VDCS and is improved in phase II to simulate the control plant. A network simulation interface is implemented that interfaces the network devices and control plant to exchange control signals and plant operating states. The plant simulation is able to integrate third-party process simulation software in the VDCS simulation environment to perform a complicated system.

4.9.1 *Plant Simulation*

A plant simulation includes a plant application simulation and plant simulation services. The plant application simulation is implemented in the Plant Simulation Component. Similar to the device application simulation, the implementation of the plant application simulation is separated from the simulation core. The Plant Simulation Component is encapsulated in a DLL; the Plant Service component remains in the simulation core. Figure 31 illustrates the implementation structure of the plant simulation and the device application simulation.

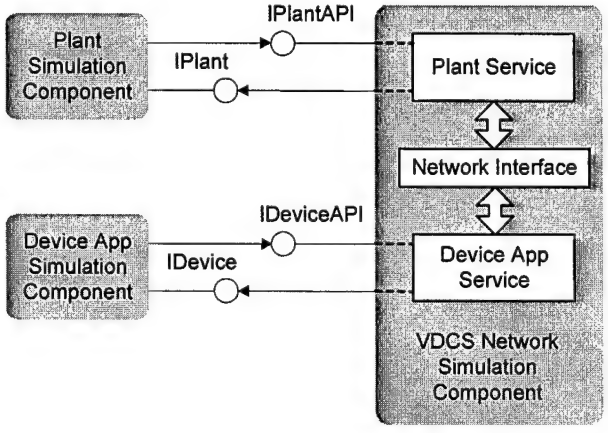


Figure 31 Plant and Network Simulation Model

Since the control plant of a control network may vary from one application to another, VDCS accepts a user defined C style pseudo-code to define the plant application. A plant application is

simulated iteratively. The plant simulation code defines a single iteration of the plant application. The simulation generator translates the simulation code and tailors it into the network simulation structure. The simulation generator uses the console msdev.exe program to build the project for the Plant Simulation Component. A project file defines the settings for the compiler to build the component. The definition of the plant application simulation is described in *Appendix II.*

The Plant Service is implemented in the working thread responsible for the plant simulation. The thread is incorporated into the simulation core and is synchronized by the simulation controller. Both of the Plant Simulation Component and the Plant Service provide interfaces to each other to perform the plant application simulation. The Plant Simulation Component has an interface IPlant. The Plant Service has an interface IPlantAPI. They are interfaces that are active during a network simulation for exchanging plant simulation states and control signals. The Plant Service also provides APIs and simulation services for the plant application simulation.

A Network Interface connects the plant simulation and device application simulation to buffer the data and signals between plant and device application. Both of the plant service and the device application service provide access functions to this interface. The implementation of the Network Interface is described in the next section.

4.9.2 *Network Simulation Interface*

The network simulation interface is the interface between the plant simulation and the device application simulation. The network simulation interface is an array of buffers that hold data from the plant simulation or the device application simulation. The network simulation interface employs two sets of data exchange APIs: one for the plant simulation and one for the device application simulation. A connection between a plant and a device IO is represented by the two references to the same network simulation interface buffer in a plant simulation and a device application simulation respectively. Figure 32 illustrates two such connections between plants and devices.

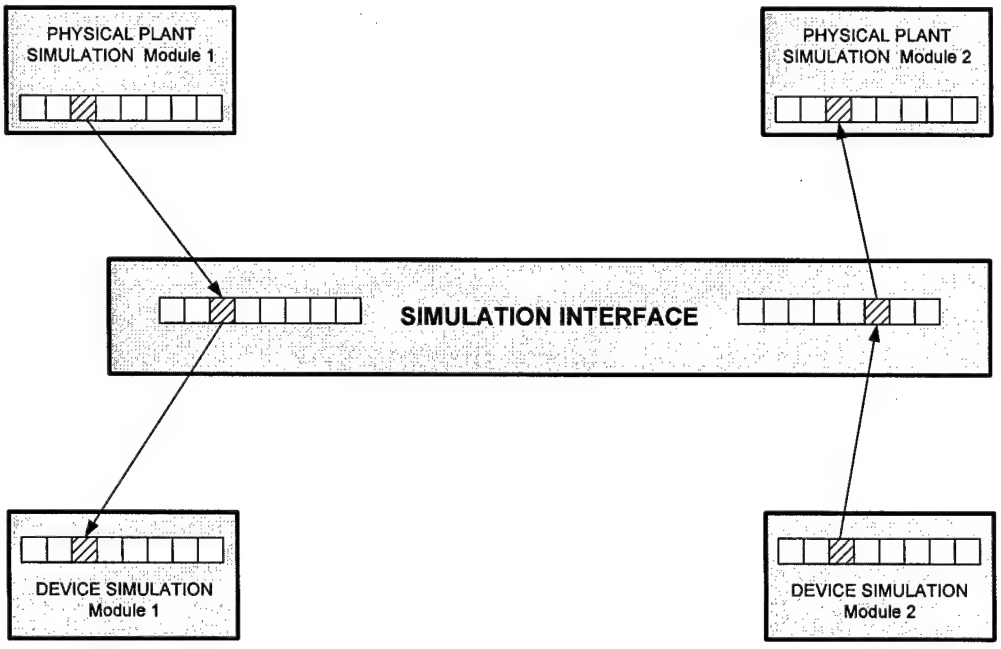


Figure 32 Network Simulation Interface

As shown in the figure, the third data element in Plant Simulation 1 represents a plant output. It updates the third network simulation interface buffer. The third I/O object of Device 1 at the same time is updated according to the same simulation interface buffer. Therefore the third I/O of Device 1 is updated with the value of the third variable of Plant 1 simulation. Similarly, the third I/O of Plant 2 simulation is updated with value of the third I/O of Device 2.

4.9.3 *Third-Party Simulation Tool Integration*

A key objective in the development of VDCS was to integrate the tool with other third-party simulation and visualization tools. Fairmount Automation had independently developed a tool that functions as a simulation manager coordinating the execution of various stand-alone simulation software packages. Using this tool, we are able to set up a system simulation with the VDCS test platform simulating the control network and a third party process simulation tool (FlowMaster for example) simulating a physical control plant. We also developed a method to implement a network management device simulation that interfaces HMI software, such as Wonderware InTouch.

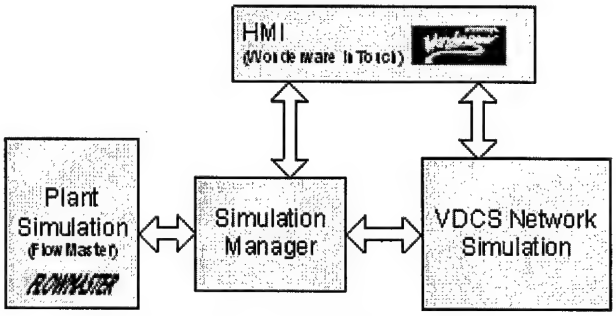


Figure 33 System Simulation Archtecture

Figure 33 illustrates the overall simulation architecture. The Simulation Manager activates and drives the Plant Simulation. We customized the Simulation Manager by defining a VDCS client and server for data exchange. The plant simulation in the VDCS Network Simulation object drives the Simulation Manager module to run the simulation. Data from the control network simulated in the VDCS Network Simulation module is reported to the HMI module via a network management device, which is typical for a network application. The network management device sends network states and values to the HMI and receives instructions from the HMI using a DDE server defined in the HMI software. The simulation manager exchanges data and signals with the HMI in the same way. The plant simulation data from Plant Simulation module and externals commands from HMI module are tunneled through the simulation manager. Detailed implementation of this architecture is explained in section 4.11, where a validation test is described.

4.10 Device and Channel Statistics

VDCS includes analysis tools to evaluate network performance. These analysis tools are designed to display the network performance in different levels of detail, depending on the interests of a

network developer. The analysis tools measure network performance with various statistics of network simulation parameters, such as the channel throughput, collision rate, etc. These statistics are computed within the simulation core both at the channel level and at the device level at a fixed update interval.

The channel level statistics monitor the channel traffic including the total count of transmitted packets, the channel throughput, collision count, collision rate, average packet size, bandwidth utilization, and message counts according to their service type.

The device level statistics monitor device activity in terms of outgoing messages. The device operation status and its impact on the channel traffic are the major focus. The statistics include the count of total transmitted messages, the count of scheduled messages, the time used for transmission, and the bandwidth utilization of the device.

Each analysis tool is associated with a set of simulation statistics. The relevant network statistics are computed only when the network analysis tool is enabled to improve performance. However, there is a set of fundamental statistics that all other statistics are based upon that are computed whenever a simulation is executing. These fundamental statistics include simulation time, channel transmitted message count, total message size, collision count, total channel transmission time, device transmitted message count, device transmission time, and message count according to different service types.

4.11 Integrating the Network Simulator with Third-Party Simulation Tools: A Practical Example

We tested **VDCS with a fluid control system** involving 8 network devices, a third party plant simulation tool, and two human-machine interfaces (HMI). The test represented a likely future real world system simulation scenario that VDCS will be called upon to execute. The demonstration platform consists of a virtual piping system implemented in a FlowMaster fluid system simulation integrated with a Smart Valve model simulation that is controlled by a real (hardware-in-the-loop) LONWORKS network. An illustration of the simulated piping system is shown in Figure 34. The simulated piping system has 3 nozzles, 9 valves, 2 pumps and 1 tank. There are six valves in the rectangular area that form two zones around the T intersections in the piping network. Each set of three Smart Valves in a zone try to detect a rupture within a zone and isolate the zone to avoid fluid loss and pressure drops in the pipes. The three Smart Valves apply a rupture detection algorithm and control logic to perform this task. Valves are also used in the supply pipes at the two ends of the pipe network. LonTalk devices are employed for Smart Valve communication to exchange flow rates.

Figure 35 is a logical representation of the piping control system of the demonstration platform. The control system shown in blue applies a LONWORKS control network with 8 devices to control the system in automatic mode.

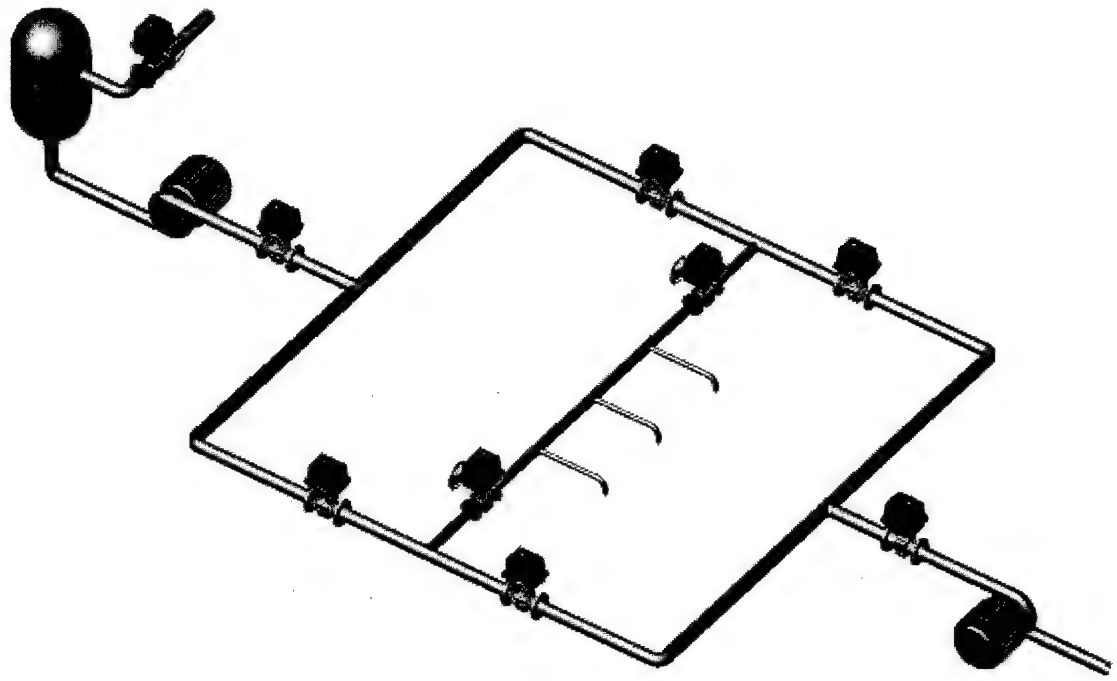


Figure 34 An illustration of the Physical Piping System with Smart Valves

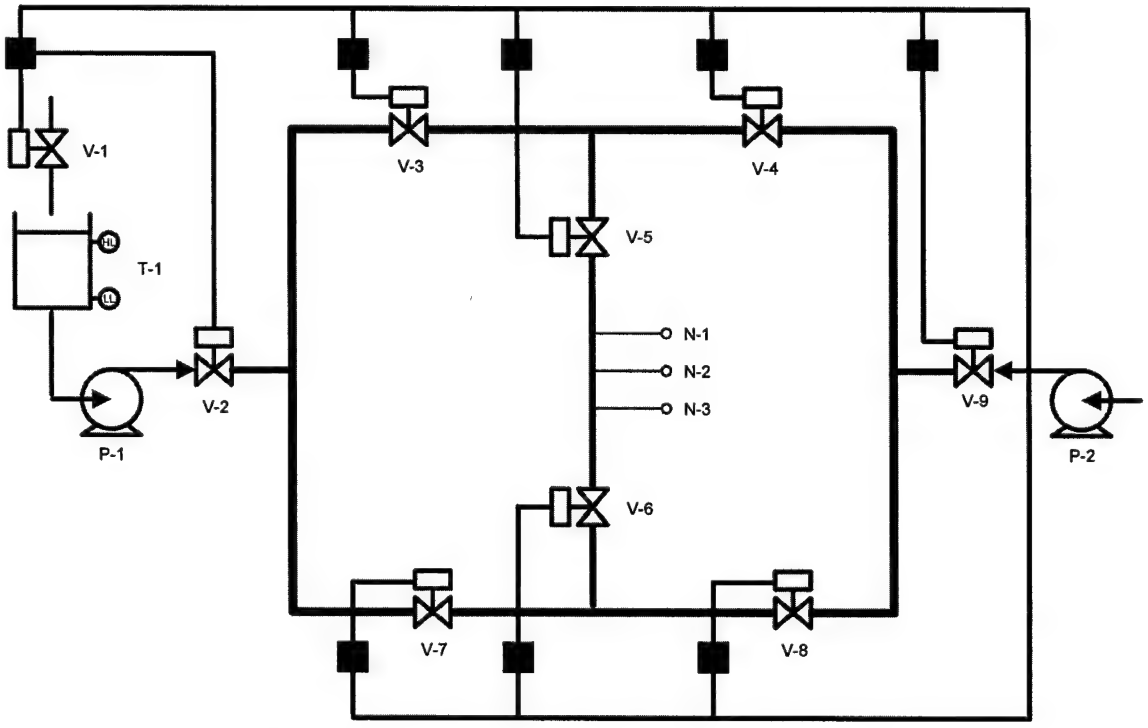


Figure 35 Piping System and Control Network Schema

The FlowMaster simulation simulates the fluid dynamics of the piping system including the pipes, valves and pumps. The VDCS full-scale network simulation tool replaces the control network hardware in the physical demonstration platform and simulates the control network dynamics. A simulation manager exchanges data between the FlowMaster simulation and the network simulation.

Two HMIs are designed using Wonderware InTouch displayed in two PCs to monitor and control the piping system and the control system respectively. The two HMIs are modified and combined into one HMI with two monitor windows that can be switched back and forth. The full-scale simulation software provides simulation analysis tools like Network Analyzer and Application Monitor to observe the channel traffic, the piping system performance and the control system control signals.

Figure 36 shows a snapshot of the simulation. The background is the control system HMI indicating a rupture detected in the zone as highlighted by a red arrow. A Network Analyzer on the upper right side monitors the channel performance. An Application Monitor on the left monitors some critical pressure readings in the piping system. The part of the full-scale simulation software main window is at the lower right corner.

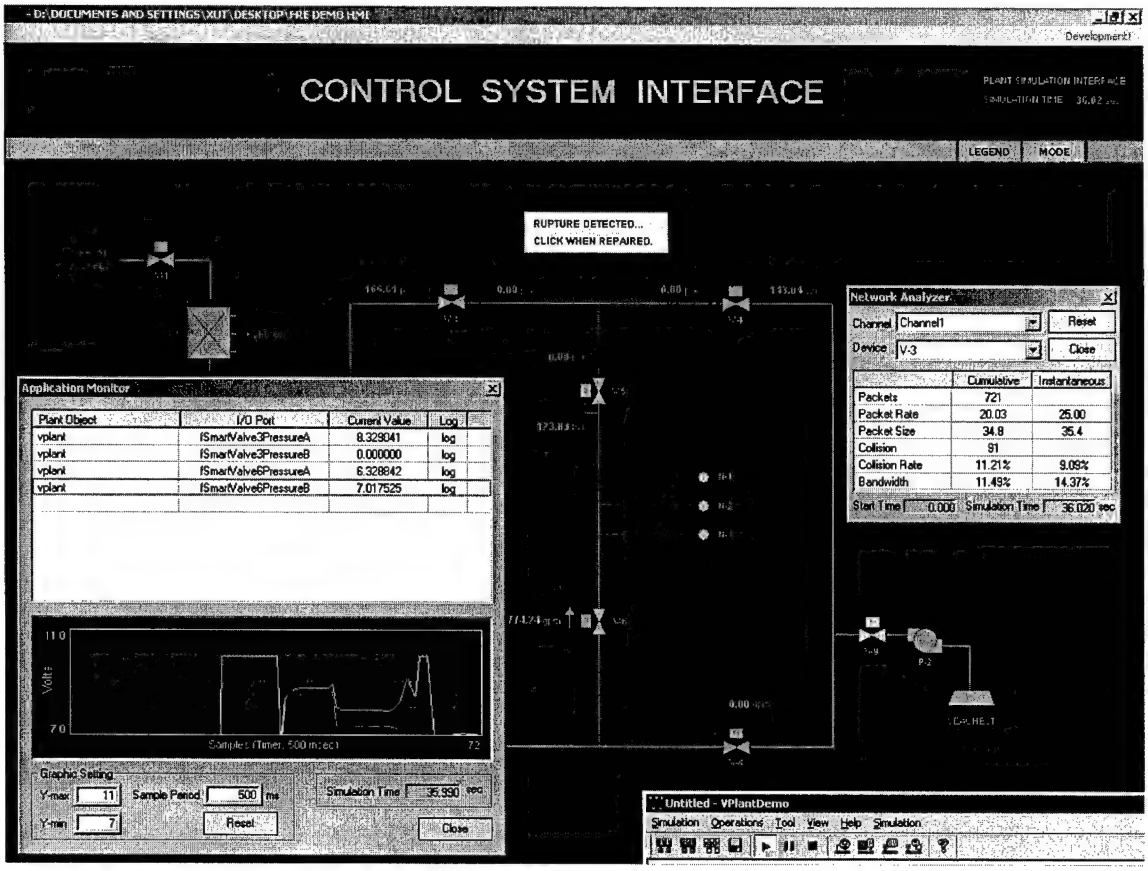


Figure 36 ASNE Demo Snapshot

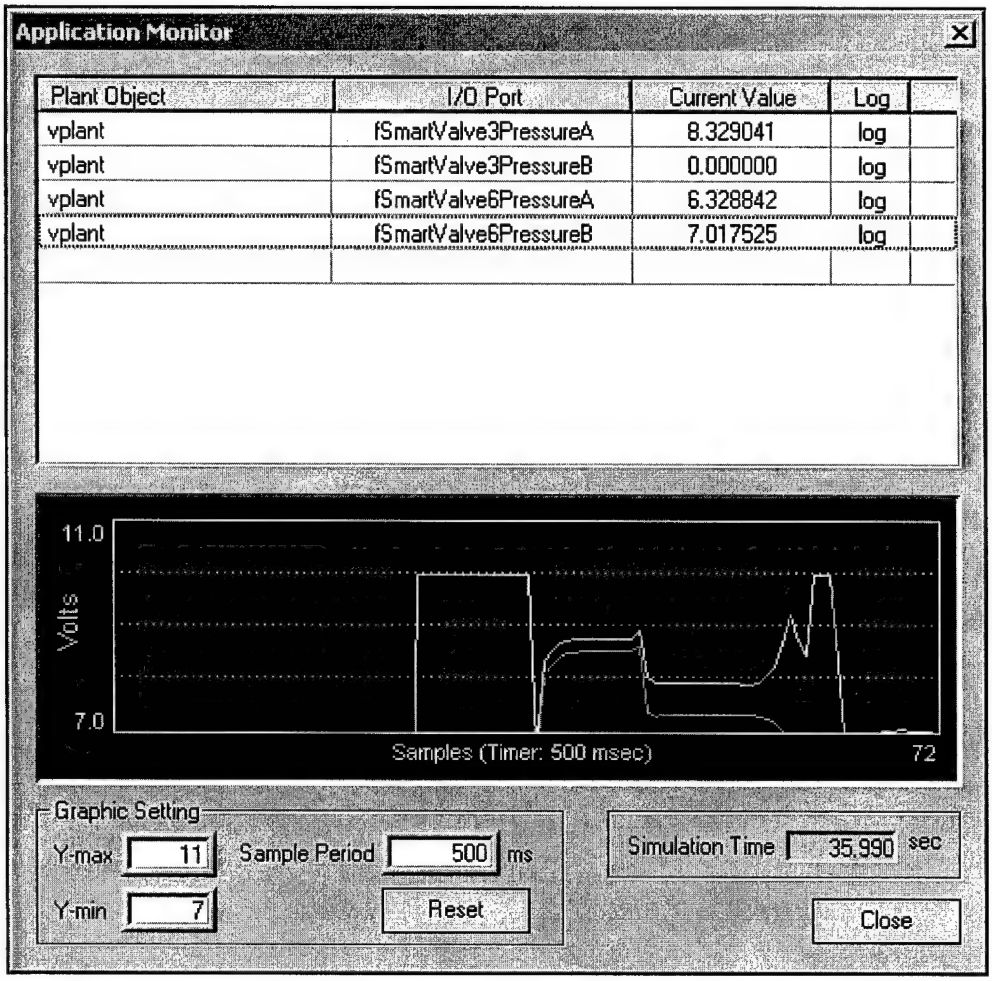


Figure 37 Application Monitor with Graphic

Figure 37 is a zoom-in view of the Application Monitor window. The Application Monitor is a simulation analysis tool built into VDCS that monitors the I/O signals of both the network device and the plant simulations. The I/Os of interests can be selected in this dialog and the value of each I/O is updated periodically according to the data value in the network simulation interface.

5. DESIGN TOOL USING THE NETWORK SIMULATOR

VDCS provides an integrated platform to simulate both the dynamics of the control network and the physical system under control. In this chapter we describe a design tool based on the Network Simulator and explain how it simulates network channel traffic and evaluates the network messaging schemes for a LONWORKS network design. The software is designed to be easy to use, fast and user friendly, while also providing very accurate results. The tool provides the same network traffic statistics that a Protocol Analyzer connected to a physical system would provide. It also generates additional statistics on network traffic that are very useful to a control system designer and that are not available by any other means (i.e., protocol analyzers do not provide this information). Appendix III provides a User Guide to this software and describes all of the features included with the tool. This chapter describes the software architecture and implementation details.

5.1 Software Architecture

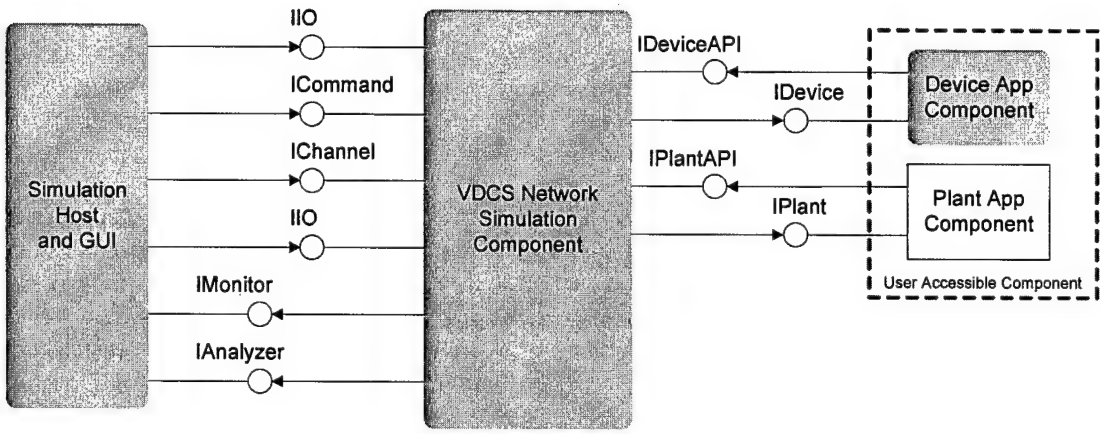


Figure 38 Network Simulation Tool Architecture in Design Tool

Figure 38 shows the architecture of the network simulation tool in the Design Tool. The VDCS Network Simulation Component, which is the simulation kernel, remains the same as in the full-scale tool. The Device Application Component and Plant Application Component also remain unchanged, although the plant simulation is not used in the Design Tool (the network analysis tool components are removed in this case.) The GUIs including the analysis tools are integrated into the Simulation Host.

Since the plant is not simulated in the Design Tool, the plant simulation and the network/plant interface are no longer necessary in the Simulation Generator, which results in a simplified structure. The architecture for the simplified Simulation Generator is shown in Figure 39. The Network Simulation Definition only depends on the LNS Database and the device application source code. Therefore user inputs are dramatically reduced.

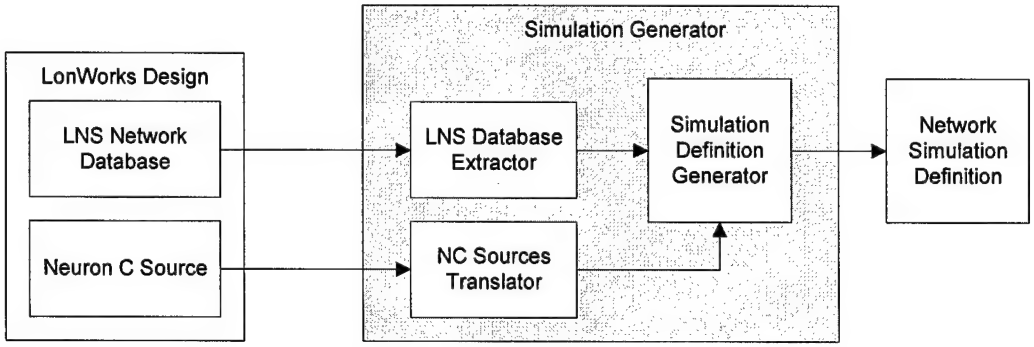


Figure 39 Simulation Generator Architecture in Design Tool

The Design Tool seamlessly integrates the Network Simulation Tool and the Simulation Generator into a single software package. The integration of all the software applications is explained in the next section. The *Design Tool User Guide* in Appendix III also provides a detailed description for how to use this package.

5.2 GUI and Application Integration

The Design Tool contains all of the simulation facilities including the Simulation Core dynamic link library, the Device Application simulation dynamic link library, the Simulation Generator application, the NC2C translator application and other supporting applications. A user GUI was designed to provide a user interface and to manage all the facilities that support the simulation. The simulation tool also depends on the LNS database for network configuration and on the Microsoft MSDEV application to build the Device Application simulation dynamic link library.

Figure 40 shows the software architecture and the relationship between the software and the supporting components and applications. We named the main application for the network simulation *LonSim*. From Figure 40, we can see that the GUI launches Simulation Core together with Device Application to perform a network simulation. The GUI retrieves the statistical data from the Simulation Core for display.

When a new simulation is built, the GUI starts the Simulation Generator. The Simulation Generator accesses the LNS network database and uses the NC2C Translator and MSDEV application to build a new simulation executable. The simulation configuration, which is accessible by the Simulation Core, is temporally stored in the machine and can be saved for future use.

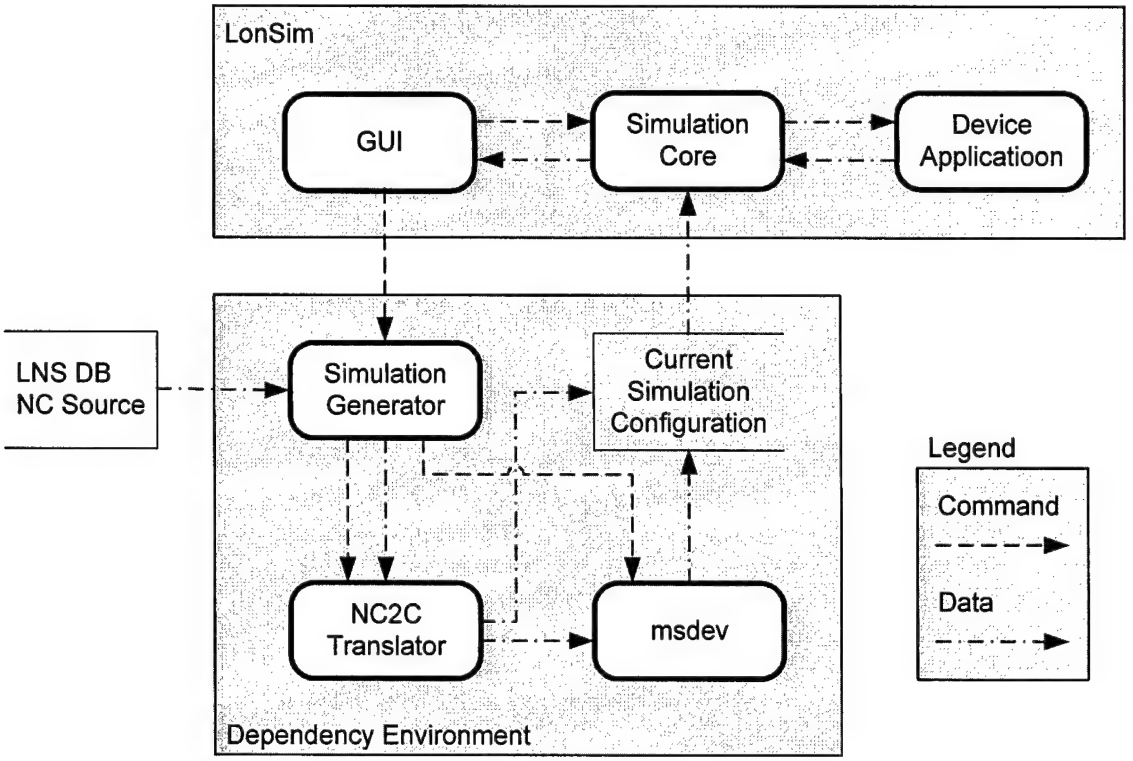


Figure 40 Software Architecture and Dependency

The additional dependency environment is the folder and files hierarchy designed as shown in Figure 41. The software is installed in the root directory named LonSim. In the LonSim folder there are five sub-folders: bin, Common, DeviceApp, DLLs, and Network. The bin folder contains the applications including format.exe, LonSim.exe, nc2c_beta.exe, VDCS_Gen.exe, and wSpawn.exe. The network configuration files used for network simulation are also stored temporally in this folder. The common folder contains common files used by multiple modules. All the dynamic link libraries and debugging related files such as the program database file are located in the DLLs folder. The files associated with the Device Application simulation project is saved in DeviceApp folder. The temporally saved network configuration files can be saved in the network folder. The sim.fmt file in this folder maintains a list of the saved network simulations.

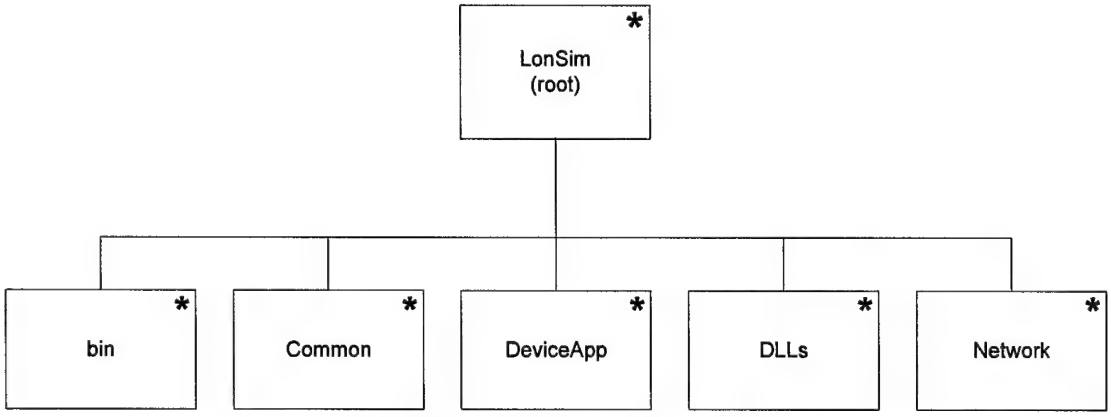


Figure 41 Design Tool Folder Hierarchy

The GUI uses a message window to display operation information of the simulation tool. To uniformly display the output of different applications, we employ memory pipes to propagate output from secondary applications to the main application. Figure 42 shows the scheme that is used to propagate the application outputs. A pipe is used between LonSim and VDCS_Gen to forward messages to be displayed. There is a second pipe that propagates outputs from NC2C and MSDEV to VDCS_Gen. VDCS_Gen propagates its own output messages and the received message to LonSim through the first pipe. All the output messages are displayed in the message pane in LonSim.

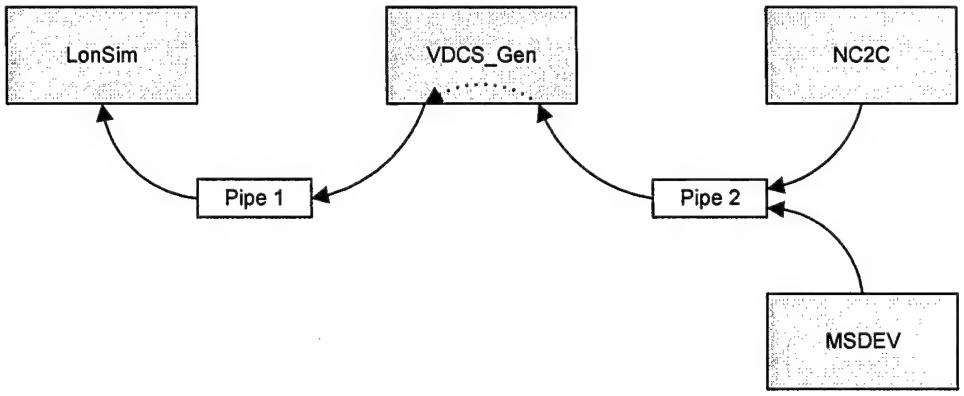


Figure 42 Message Propagation with Memory Pipes.

During a network simulation build process, the main window constantly checks the progress of the simulation building process and displays progress information from VDCS_Gen.exe. To improve the user interface, we added a working thread to take care of this heavy working load of message processing in order to avoid a less responsive user interface.

5.3 Analysis Tools

To identify the critical features a design tool must have and to increase its analysis abilities, we surveyed a number of commercial products and technical papers for network design and simulation. Three products were identified that have features applicable to the design tool. These commercial tools are CAN Open Environment (CANoe) from Vector Informatik, Germany; LOYTEC Protocol Analyzer (LPA) from LOYTEC, Germany; and LonManager Protocol Analyzer from Echelon, USA. The CANoe is the design tool for CAN networks. This product is very similar to our design tool. The other two tools are protocol analyzers with similar features. They are tools designed for design analysis and trouble-shooting.

A number of features were identified in our survey as critical ones that should be implemented in the Design Tool. These features are listed next.

- NV configuration and display in format.

  NV configuration is used to configure the device by set configurable NVs. It provides the flexibility to modify the device behavior with a fixed application. This feature also can reduce the number of network applications in certain cases.

- Device message loss ratio.

  This feature shows the number of messages lost within a device. When messages are lost, it implies that the device scheduled more messages than it can handle or that the channel would allow going through. This feature is only available in a simulation.

- Device bandwidth utilization and contribution to total bandwidth utilization.

  This feature shows the bandwidth utilization of an individual device and it's contribution to the total channel bandwidth utilization. With this feature, it is very easy to identify the most vulnerable device by comparing the bandwidth utilization of all the devices in a channel.

- NV log and view (in raw data format)

  This feature logs and display transmitted network variable including it's time stamp, modes, addressing and data. This feature may be used to monitor and trace individual messages. It also shows the total time delay for a routed message to be received.

- Channel traffic load monitoring and comparison.

  This feature provides the basic evaluation of channel performance and makes it easy to identify overloaded channels. It will help the user to balance the traffic throughout of the network.

- Router load monitoring and comparison.

  This feature reveals the router packet load to help to identify the improper design of the inter-channel connections and the most vulnerable routers. It is a valuable feature to improve the overall network performance.

Various network analysis tools are developed for VDCS according to the list. The *Design Tool User Guide* included in Appendix III provides a comprehensive description of these analysis tools. There is another analysis tool called Application Monitor developed in the full-scale simulation tool for monitoring the I/O signals of device and plant simulations.

5.4 Device Application Debugging

The device application debugging function is another promising feature of the VDCS simulation tool, which may be the most appealing trouble-shooting feature that VDCS offers. Debugging is an essential method in software development and trouble-shooting. However, the networking application running on devices over a network imposes additional difficulties in debugging. It is critical to trace the response of each device to others. Unfortunately, the LONWORKS network design tool LonBuilder only offers the ability to debug device application on one individual device

at a time. During a debugging session only the application on the debugged device is under control. The applications on other devices just run at their normal speed. The interaction between devices under normal operating conditions is not possible to repeat in a debugging session. Therefore the external interaction of device application in a debugging session is not equivalent to the scenario of a normal operation, which very much limits the debugging ability.

On the other hand, the debugging of the network simulation is a promising approach to debug the device applications supposing the network simulation is able to accurately represent the network dynamics. Since the network simulation is pure software itself, many sophisticated software program debuggers are available and can be directly applied to debug the device applications. Since the simulation tool is developed on VC++ IDE, by default we use the VC++ debugger to debug the device application and plant simulation.

According to the simulation generation procedure, the user needs to compile and build the simulation project in the final step before running the simulation. It is advisable not to expose a user to the network simulation details such as the protocol simulation, channel communication, thread synchronization, graphic interface, etc. Making the simulation code invisible for the user will help him/her concentrate on the user applications in the event of application code debugging. We applied COM technology to the VDCS in order to improve the application program debugging ability. Both the user application simulation and network protocol simulation are separated and encapsulated into DLL COM components. User can debug one DLL without knowing the tedious source code in other DLLs. This approach makes the network simulation transparent to the user during a device application debugging session. A formalized procedure to debug the device application is described in *Design Tool User Guide* (Appendix III). The plant simulation may be debugged in a similar way using the full-scale simulation tool.

6. VDCS VALIDATION

In VDCS project phase II, we validated the VDCS design with the Chilled Water Reduced-Scale Advanced Demonstrator (CW-RSAD), and verified the simulation accuracy of channel traffic and device applications. The CW-RSAD is an advance demonstrator that employs a medium-scale LONWORKS network with over 100 network devices to control a piping system to provide chilled water for cooling. The LONWORKS network on the CW-RSAD has 16 channels including an Ethernet channel. The network consists of 108 network devices and 11 LonPoint routers and 4 iLon gateways. Under normal operation, when the piping system reaches a steady state, each network device propagates its sampled signals at a constant rate (one message every 5 seconds). Measurements of channel throughput, bandwidth utilization, and average packet message size under normal system operation were gathered from the CW-RSAD network.

The network design was re-generated for this testing effort, as is shown in Figure 43, A simplified version of device applications were developed to perform the task of normal network operation. In the test, all the devices sent a message every 5 sec for each network variable to its destination. All the messages were also fed to a network management device. Since the network management device on an Ethernet channel cannot be easily simulated, an extra channel was introduced to receive all messages. Since there no messages come out of the network management tool, this modification will not compromise the accuracy of the network simulation.

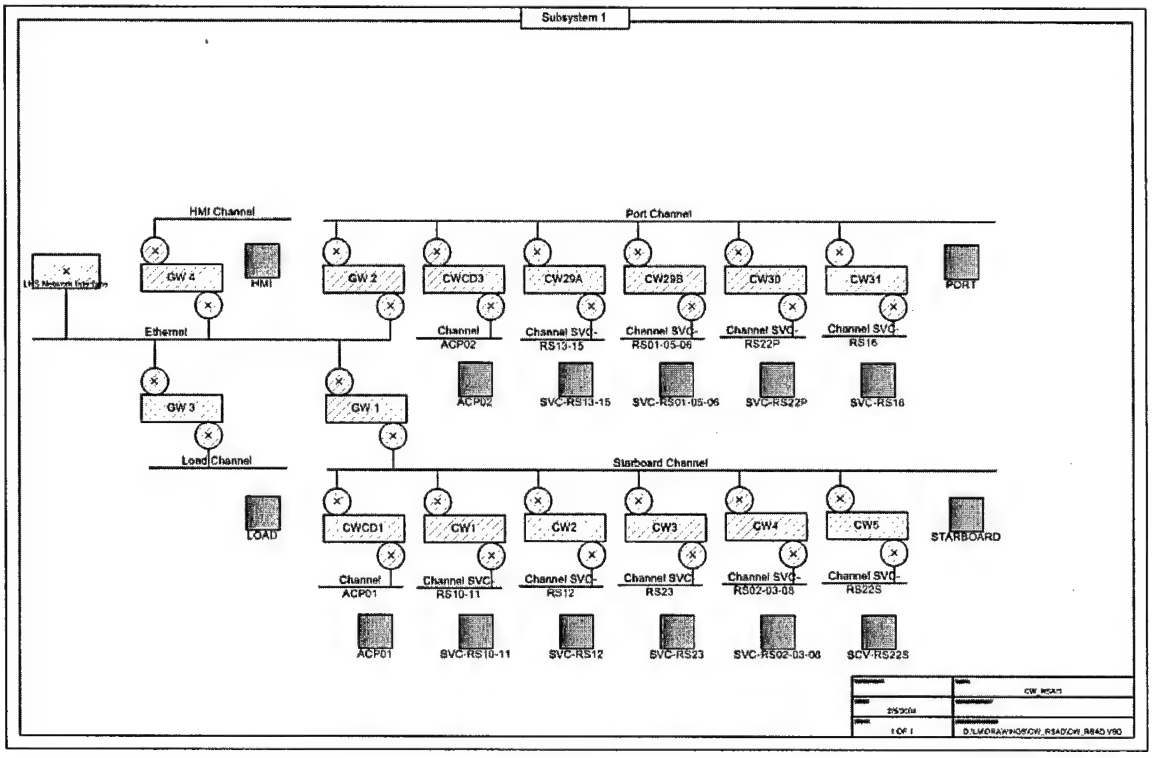


Figure 43 RSAD Network Design in LonMaker

The data gathered in the VDCS simulation and the RSAD system are compared in Table 5. Three channel statistical parameters are compared for each communication channel, namely the channel throughput, the bandwidth utilization and the average packet size. Since the network throughput and average packet size are predictable, the expected result is also listed in Table 5. The data of the RSAD system and the expected result are classified and highlighted according to the degree of agreement with the corresponding VDCS simulation data. A white cell means the data matches closely to the simulation data. A gray cell indicates the data differs slightly from the simulation data. An orange cell indicates that the simulation data do not agree with the data in the cell.

Table 5 Comparison of Network Traffic Statistics

	RSAD system			VDCS simulation			Expected Result		
Channel	Pkt/sec	BW	Size	Pkt/sec	BW	Size	Pkt/sec	BW	Size
PORT	15.3	9.0%	26.4	15.5	7.1%	26.9	15.8	n/a	26.5
STARBOARD	21.4	13.0%	27.8	15.5	7.2%	26.8	15.8	n/a	26.4
LOAD	13.7	8.0%	23.2	11.7	5.1%	23.8	12.0	n/a	22.5
ACP01	4.4	3.0%	25.6	3.8	1.8%	25.4	3.8	n/a	26.0
SVC_RS10-11	2.0	1.0%	23.9	2.0	0.9%	25.5	2.0	n/a	24.4
SVC-RS12	0.9	<1.0%	24.6	1.0	0.5%	25.3	1.0	n/a	24.4
SVC-RS23	1.4	<1.0%	26.1	1.4	0.7%	27.4	1.4	n/a	26.6
SVC-RS02-03	4.2	2.0%	26.5	4.1	1.9%	27.3	4.2	n/a	26.6
SVC-RS22S	1.4	<1.0%	26.1	1.4	0.7%	27.4	1.4	n/a	26.6
ACP02	4.6	3.0%	25.9	3.8	1.8%	25.4	3.8	n/a	26.0
SVC-RS13-15	3.5	2.0%	24.8	2.9	1.3%	25.5	3.0	n/a	24.4
SVC-RS01-05	4.0	2.0%	26.9	4.1	2.0%	27.3	4.2	n/a	26.6
SVC-RS22P	1.3	<1.0%	26.9	1.4	0.7%	29.3	1.4	n/a	26.6
SVC-RS16	3.7	2.0%	17.4	1.4	0.7%	27.8	1.4	n/a	26.6

Table 5 shows that the channel throughput in the VDCS simulation matches the data of the expected result closely. The average packet size in the simulation also matches the expected result with small disagreements in two cases. Comparison of the simulation data and the operation data from RSAD system, including the bandwidth utilization, indicates that the performance of the hardware and the simulation are very close in most cases. For the STARBOARD channel and the SVC_RS16 channel, the hardware data do not agree with the simulation data and the expected result. The difference is due to the abnormal operation of a few network devices in those two channels. In normal operation, a network device heartbeats messages at a constant speed. However, if some sensor signals connected to the device I/Os changes frequently, the device will generate more traffic. Although the hardware data was gathered at the normal operation state, the oscillation of certain sensor signals forced these device sto send more messages than they do in a normal state. With the exception of these anomalies our simulations provided excellent predictions for the network designer.

7. THE COMMUNICATION SIMULATOR

In this chapter, we describe the communication simulator, a data network modeling tool that we have developed for LONWORKS distributed control networks. The modeling tool, using OPNET Modeler®, allows multiple system configurations to be defined and examined completely. The Communication Simulator assumes more *a priori* information than the Network Simulator. On the other hand, it is much quicker than the network simulator, since it does not require that all control algorithms run simultaneously with the communication infrastructure simulation. This chapter consists of four parts.

In section 7.1, we introduce the implementation details of the network models by using OPNET, which is one of the most popular engineering tools used for data networking research.

In section 7.2, we described a modeling tool we developed for LONWORKS communication protocol and its detailed implementation features, architecture, interface and applications.

In section 7.3, we discuss two analytical models for LONWORKS protocol. The first one studies the behavior of the channel from the "channel viewpoint", while the other takes the "node viewpoint", focusing on the MAC layer (predictive p-persistent CSMA algorithm). The analytical model we provided for the node uses a Markov chain model, under the key assumption that the collision probability of a packet transmitted by each node connected to the LONWORKS communication channel is constant and independent of other collisions.

Finally, we verify the performance of the modeling tool by comparing results gathered from simulations, from a physical system using LONWORKS nodes, and from the analytical models (section 7.4).

7.1 Background - OPNET Modeler Description

The OPNET (Optimized Network Engineering Tool) Modeler [11] was first demonstrated at MIT in 1987. It is a comprehensive software environment for modeling, simulating, and analyzing the performance of communications networks and protocols, computer systems and applications, and distributed systems. OPNET Modeler contains many models of existing hardware, allowing quick construction, test and analysis of many types of networks and protocols including from a single LAN to global satellite networks. Discrete event simulations are used as the means of analyzing system performance and behavior. The key features of OPNET are summarized as follows:

Object orientation
Systems specified in OPNET consist of objects, each with configurable sets of attributes. Objects belong to "classes" which provide them with characteristics in terms of behavior and capability. Definition of new classes is supported in order to address as wide a scope of systems as possible. Classes can also be derived from other classes, or "specialized", in order to provide more specific support for particular applications.

Specialized in communication networks and information systems
OPNET provides many constructs relating to communications and information processing, allowing high leverage for modeling of networks and distributed systems.

Hierarchical models
OPNET models are hierarchical, naturally paralleling the structure of actual communication networks.

Graphical specification
Wherever possible, models are entered via graphical editors. These editors provide an intuitive mapping from the modeled system to the OPNET model specification.

Flexibility to develop detailed custom models
OPNET provides a flexible, high-level programming language with extensive support for communications and distributed systems. This environment allows realistic modeling of all communications protocols, algorithms, and transmission technologies.

Automatic generation of simulations
Model specifications are automatically compiled into executable, efficient, discrete-event simulations implemented in the C programming language. Advanced simulation construction and configuration techniques minimize compilation.

Application-specific statistics.
OPNET provides numerous built-in performance statistics that can be automatically collected during simulations. In addition, modelers can augment this set with new application-specific statistics that are computed by user-defined processes.

Integrated post-simulation analysis tools
Performance evaluation, and trade-off analysis require large volumes of simulation results to be interpreted. OPNET includes a sophisticated tool for graphical presentation and processing of simulation output.

Interactive analysis
All OPNET simulations automatically incorporate support for analysis via a sophisticated interactive "debugger".

Application program interface (API)
As an alternative to graphical specification, OPNET models and data files may be specified via a programmatic interface. This is useful for automatic generation of models or to allow OPNET to be tightly integrated with other tools.

OPNET comes complete with a range of tools that allows developers to specify models in great detail, identify the elements of the model of interest, execute the simulation and analyze the generated output data. It offers four editors at different component levels to help users to develop a representation of a system being modeled. These four editors, Network, Node, Process and parameter editors are organized in a hierarchical way, which supports the concepts of model level reuse. Models developed at one layer can be used by another model at a higher layer. For instance, the Network Editor makes use of node and link objects; the Node Editor provides processors, queues, transmitters, and receivers; and the Process Editor is based on states and transitions. Figure 44 shows this hierarchical structure. For the purpose of better understanding the modeling tool we developed for the LONWORKS system, we introduce each of the editors in details. The parameter editor is always seen as a utility editor, and not considered as a modeling domain.

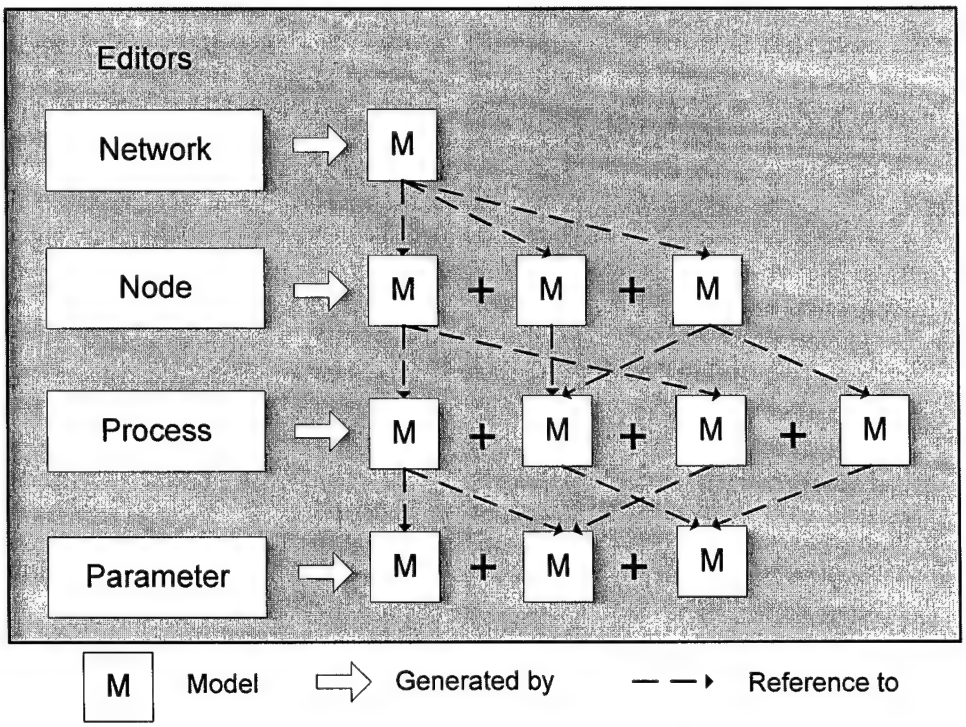


Figure 44 Hierarchical Organization of Editors

Network Editor

The Network Editor is used to specify the physical topology of a communications network, which defines the position and interconnection of communicating entities, i.e., *node* and *link*. The specific capabilities of each node are realized in the underlying *model*. A set of parameters or characteristics is attached with each model that can be set to customize the node's behavior. A node can be either fixed, mobile or satellite. Simplex (unidirectional) or duplex (bi-directional) point-to-point links connect pairs of nodes. A *bus link* provides a broadcast medium for an arbitrary number of attached devices. Mobile communication is supported by *radio links*. Links can also be customized to simulate the actual communication channels.

The complexity of a network model would be unmanageable if all networks were modeled as part of a single-level system. This complexity is mitigated by an abstraction known as a *subnetwork*. A network may contain many subnetworks. At the lowest level, a subnetwork is composed only of nodes and links. Communications links facilitate communication between subnetworks.

Node Editor

Communication devices created and interconnected at the network level need to be specified in the node domain using the Node Editor. Node models are expressed as interconnected *modules*. These modules can be grouped into two distinct categories. The first set is modules that have predefined characteristics and a set of built-in parameters. Examples are packet generators, point-to-point transmitters, and radio receivers. The

second group contains highly programmable modules. These modules, referred to as *processors* and *queues*, rely on process model specifications.

Each node is described by a block structured data flow diagram. Each programmable block in a Node Model has its functionality defined by a Process Model. Modules are interconnected by either *packet streams* or *statistic wires*. Packets are transferred between modules using packet streams. Statistic wires could be used to convey numeric signals.

Process Editor

Process models, created using the process editor, are used to describe the logic flow and behavior of processor and queue modules. Communication between processes is supported by *interrupts*. Process models are expressed in a language called Proto-C, which consists of state transition diagrams (STDs), a library of kernel procedures, and the standard C programming language. The OPNET Process Editor uses a state-transition diagram approach to support specification of any type of protocol, resource, application, algorithm, or queuing policy. States and transitions graphically define the progression of a process in response to events. Within each state, general logic can be specified using a library of predefined functions and even the full flexibility of the C language. A process may create new processes (*child processes*) to perform sub-tasks and thus is called the *parent process.*

7.2 Implementation of LONWORKS Networks Modeling Tool

In this part, we describe how OPNET Modeler was used to model the behavior of the LONWORKS communication channel in an environment of simulated operation. We will describe the method used, the applicability to the LONWORKS communication standard, the data and functionality available, and the limitations of the model.

7.2.1 Background and Operational Description of LONWORKS protocol

The LONWORKS protocol, also known as the LonTalk protocol and the ANSI/EIA 709.1 Control Networking Standard, are the basic building blocks of the LONWORKS distributed control system. The protocol is a layered, packet-based, peer-to-peer communication protocol, which provides a set of communication devices that allow the applications program in a device to send and receive messages from other devices over the network without needing to know the topology of the network or the name, addresses, or functions of the devices. Of the most popular fieldbus protocols, LONWORKS protocol is the only one to provide services at all seven layers of the OSI reference model.

Under normal operation, LONWORKS offers two techniques for packet transmission:

Acknowledged Messaging: Provides for end-to-end acknowledgement. When using this kind of service, a message is sent to a device or group of up to 64 devices and individual acknowledgements are expected from each receiver. If acknowledgements are not received after the senders timeout, the sender retries transmission.

Unacknowledged messaging: Causes a message to be sent to a device or group of devices multiple times. No acknowledgements are expected. This message service has the lowest overhead and is the most popular service.

Media access: All the network protocols use a media access control (MAC) algorithm to allow devices to determine when they can safely send a packet. The LONWORKS protocol employs a sophisticated, unique MAC algorithm with collision avoidance, called the predictive p-persistent carrier sense multiple access (p-CSMA) protocol [12]. Compared with existing MAC algorithms such as IEEE 802.2, 802.3, 802.4 and 802.5, p-CSMA has excellent performance characteristics even during periods of network overload.

Other components of the LONWORKS network are described below:

Channel: A physical unit of bandwidth linking one or more communication nodes. LONWORKS protocol is medium-independent, which means that the protocol allows all devices to communicate with each other over any type of physical transport channel. However, each type of channel has different characteristics in terms of maximum numbers of connections, data-flow rate, and maximum communication distance. The most frequently used channel type is TP/FT-10, which supports 78kps data rate, 64-128 maximum devices, and 2200m maximum distance (bus topology).

Subnet: A set of nodes accessible through the same link layer protocol; a routing abstraction for a channel; EIA-709 subnets are limited to a maximum of 127 nodes.

Node: An abstraction for a physical node that represents the highest degree of address resolvability on a network. A node is identified within a subnet by its logical node identifier. A physical node may belong to one or two subnets. When a node belongs to two subnets, it is assigned one node number for each subnet to which it belongs.

Media Access: A media access control (MAC) algorithm is required by all network protocols. MAC algorithms are designed to minimize packet collision rate. LONWORKS protocol uses predictive p-persistent CSMA protocol.

Beta1: Period immediately following the end of a packet cycle. A node attempting to transmit monitors the state of the channel, and if it detects no transmission during the Beta1 period, it determines the channel is idle.

Beta2: Randomizing slots. A node wishing to transmit generates a random delay T. This delay is an integer number of randomizing slots of duration Beta2.

7.2.2 Basic Media Access Mechanism of LONWORKS

LonTalk uses a unique and also very efficient *media access control* (MAC) protocol called predictive p-persistent CSMA to allow nodes to determine when they can safely send a packet of data. A collision occurs when two or more devices attempt to send data at the same time. The LONWORKS MAC algorithm allows a channel to operate at full capacity with a minimum of collisions.

As with Ethernet, all LONWORKS nodes randomize their access to the medium over a minimum of W_{base} Beta2 slots, where W_{base} has a constant value of 16. A unique feature of LonTalk MAC protocol is that the number of available Beta2 slots is dynamically adjusted by every node, based on an estimation of expected network loading, *BL*, maintained by each node. BL varies from 1 to 63. Therefore, the number of available Beta2 slots, $BL^* W_{base}$, varies from 16 to 1008, depending on this estimation. This method allows the LonTalk protocol to minimize media access delays with a

small number of Beta2 slots during periods of light load while minimizing collisions with many Beta2 slots during periods of heavy load.

Figure 45 shows the *finite state machine* (FSM) we designed for a LONWORKS node according to the description of predictive p-CSMA MAC algorithm. For any LONWORKS node module, starting from *IDLE* state, it monitors the state of the communication channel, and determines if the channel is in an idle state. If it detects that the channel is busy, it will enter *WAIT* state. Otherwise, if it detects no transmission during the Beta 1 period, it will enter *BACKOFF* state. Nodes without a packet to transmit during the Beta 1 period will go back to *IDLE* and remain in synchronization. Next, the node generates a random delay time in the interval (0.. (W_{base}*BL)-1), where BL is an estimate of the current channel backlog. The node enters the *TRANSMIT* state if the communication channel is still idle when the random waiting time expired. If the transmission is successful, the node enters the *TX_END* state. Otherwise, it goes to *COLLISION* and *TX_ABORT* state. The whole process keeps repeating when there are more packets coming to the MAC layer and need to be transmitted. We implemented this FSM into the MAC layer of the LONWORKS modeling tool.

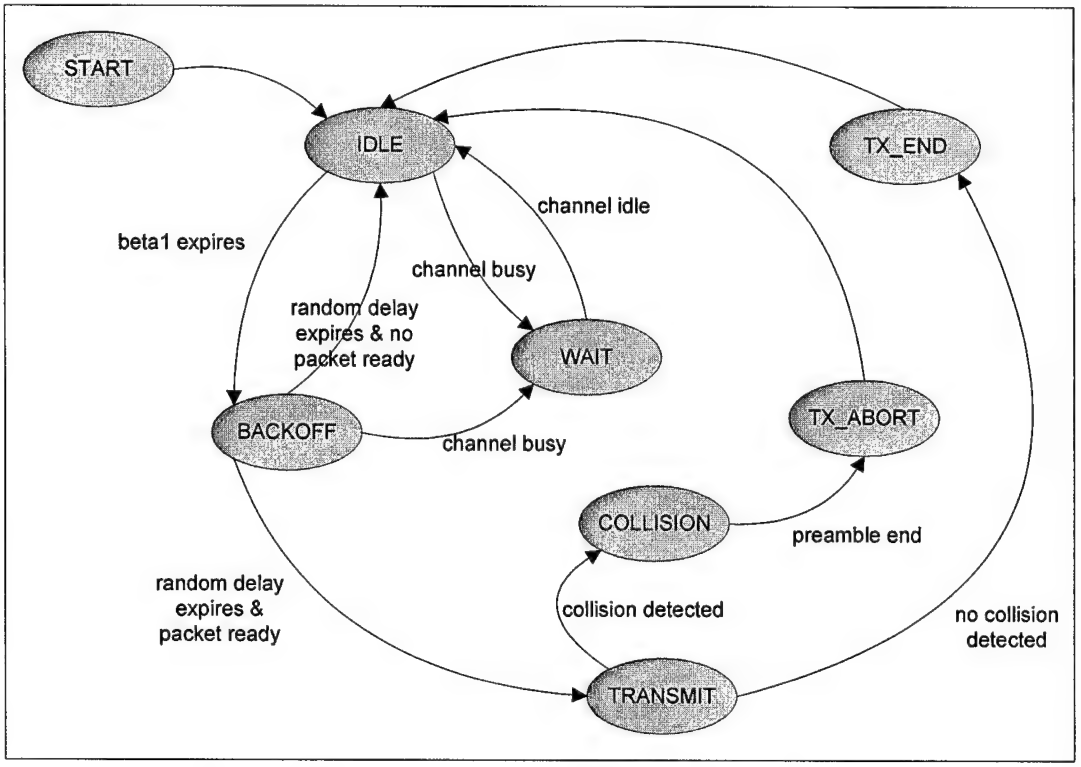


Figure 45 Finite State Machine of the LonTalk MAC Layer

Detailed information of the LonTalk protocol can be found in the documentation (ANSI/EIA-709.1-A) [10].

7.2.3 Model Scope and Limitations

This section provides an overview of the parameterized LONWORKS modeling tool implemented in the OPNET Modeler. The LONWORKS modeling tool contains a full implementation of the MAC algorithm of LonTalK protocol, Link Layer [10], and parameter setting of the physical layers.

It also contains a simulator for the LONWORKS protocol analyzer. The simulation environment offers a graphical user interface, which allows the user to form, monitor, and analyze a testing network using "drag and drop" functionality. OPNET Modeler contains libraries for the models of most standard communication technologies like Ethernet, FDDI, and ATM, which allow the user to expand a model so that it can be incorporated with a TCP/IP network via iLON 1000 (Ethernet GATEWAY). Figure 46 shows the GUI of the LONWORKS modeling tool. In the figure, we shows eight *smartcontrol*® devices along with a Protocol analyzer (LonManager) connected to the TP/FT-10 bus channel. Users can drag icons from the "Object Palette" window if they would like to add more *smartcontrol*® devices to the channel. The user can also run the simulation by clicking "run simulation" under the "Simulation" dropdown menu.

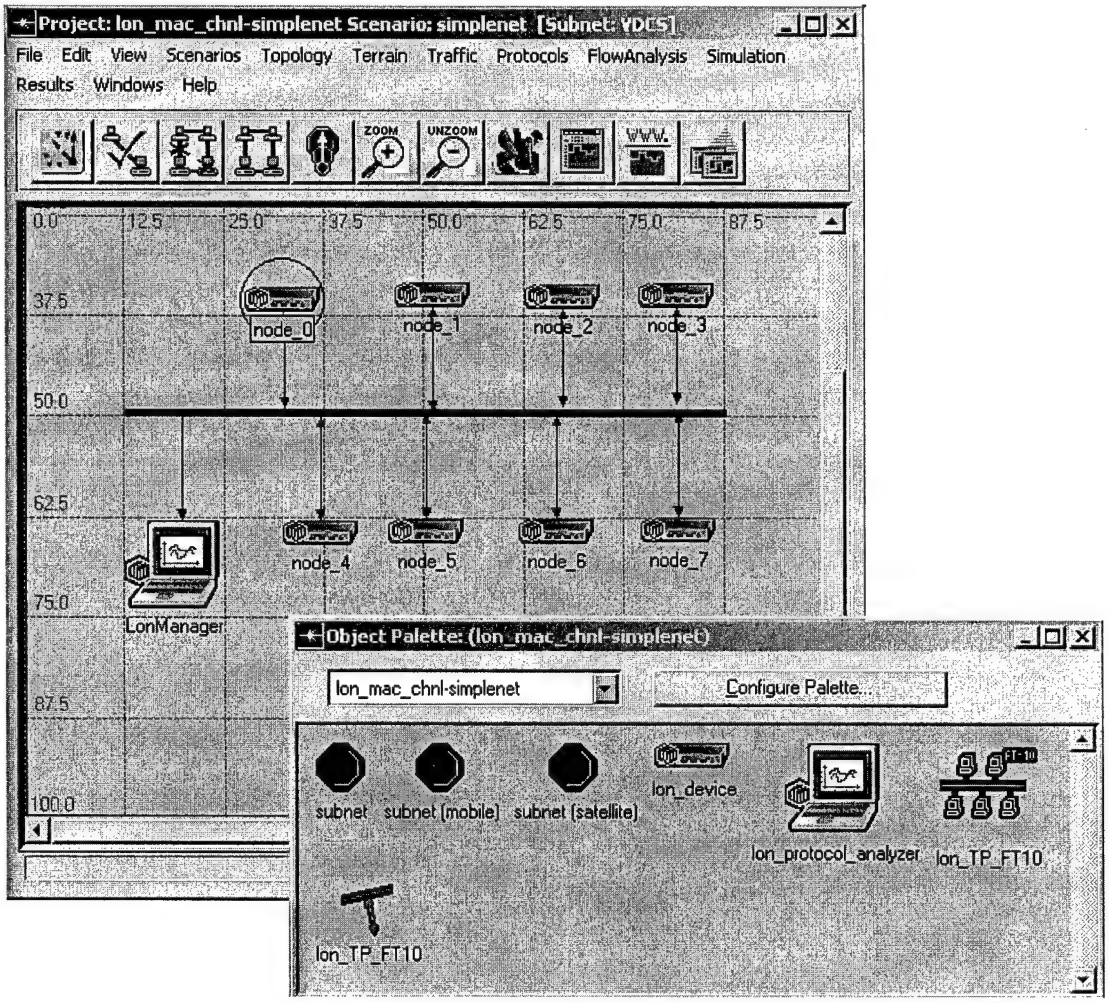


Figure 46 GUI of LONWORKS Communication Simualtor

Some parts of the protocol have been simplified, omitted or deferred for later development in view of the fact that it is intended primarily to study, and analyze the performance at the data-networking level of the communication channel. We have also reserved several APIs that we may use in future development. The following table lists the features and parameter settings we provide in the communication simulator:

LonTalk MAC Protocol Behavior Modeling	
Access Mechanism	Carrier sense multiple access with collision avoidance (CSMA/CA) and optional collision detection (CSMA/CD) access schemes as defined in the standard.
Frame Exchange	Combined with message type service provided by the Transportation Layer: acknowledged message and acknowledgment exchange for the reliability of data transfer; unacknowledged message for high-speed requirement of data transfer.
Deference & Backoff	Interframe spacing Beta1 and backoff time slot Beta2 implementation. The values are selected based on design specification from Echelon (the manufacturer of LONWORKS nodes). Random delay backlog generated from the predictive estimation of channel load BL.
Data Rate	78Kbps data rate supported by LONWORKS TP/FT-10 bus channel 1.25Mbps data rate supported by TP/XF-1250 bus channel
Input Clock	Different input clock frequencies of LONWORKS node: 0.625MHz, 1.25MHz, 2.5MHz, 5MHz, 10MHz. Different clocks have different limitations on packet rate and time slot duration.
Transmission	Full-duplex transmissions FIFO processing of transmission requests Transmission attempt limit of 256 after collision
Buffer Size	Packet arrived from a higher layer to the LonTalk MAC layer is stored in an output buffer. The buffer size is limited to the adjusted maximum value. Higher packets are dropped once the maximum buffer size is reached.
Framing & Disassembly	Assemble MAC frame before transmission, and disassemble frame when receiving from PHY layer. Frame format is defined as MPDU format specified in the standard, with additional address and message type information in order to evaluate channel performance under different message service types.
Frame Size	Frame sizes based on measurements from real LONWORKS channel are in the range of 10 to 16 bytes long.
Node Auto-addressing	All the nodes can be configured for node ID auto-addressing. Node addresses can also be specified by user. However, no subnet or group address is supported yet. Destination address of each packet can be chosen automatically from address pool of all the nodes.
Recovery Mechanisms	Retransmission mechanism for data frame based on failure of the reception of the acknowledgment frame.
Collision Detection	Collision detection from physical layer. Normal Mode or Special Purpose Mode as defined in the standard.

The communication simulator assumes that no propagation delays occur on the communication channel; only transmission delays are considered. Without the propagation delays on the communication channel, and since all stations are deferred for integral slot times of Beta2, collision

can only occur at the beginning of the transmission cycle. We assume that packet error is only caused by collisions, and that packet loss is a result of the buffer between the MAC layer and higher layers being full. In other words, the rest of the channel is ideal. That is, no bit error or packet loss occurs in the ideal channel.

It is an inherent property of technology development that as soon as a new version of hardware is released, some of the specifications are changed. Our modeling tool is heavily parameterized and thus allows significant reparameterization. A user can set physical channel bandwidth, processing speed of the newly developed Neuron chips, and buffer allocation strategies individually, for each node in the system, or once for all nodes.

7.2.4 The Structure of LONWORKS Models

7.2.4.1 Network Model

Figure 47 shows a TP/FT-10 bus network topology and the corresponding hierarchical relationship with an inside view of the node and process models that we have developed for the LONWORKS MAC model. A uniform node model that follows the LonTalk protocol is designed to process transmission and reception of packets over a bus link, as illustrated in Figure 48.

A packet generated from the source module of the sending node is transferred to the MAC process by MAC_IF (mac_interface process), being attached an ICI, Interface Control Information, which is a specific data structure for information exchange between processes in OPNET. ICI encapsulates information of increment of backlog, the destination address of the packet, and the type of the data packet. In the MAC, this information, along with the address of the source node and the original packet are assembled into a new MPDU (MAC sublayer Protocol Data Unit). This MPDU or frame will be transmitted to the channel by the transceiver of the sending node. All the nodes on the multi-access channel will receive this frame, while by extracting the destination address of the frame header, only the destination node will process this frame. Increment of backlog will be extracted to update the backlog of the destination node, and the type of packet will be sent to the upper layer MAC_IF along with the original data packet. MAC_IF will decide to arrange an acknowledgement according to the type of the received packet, acknowledged or unacknowledged.

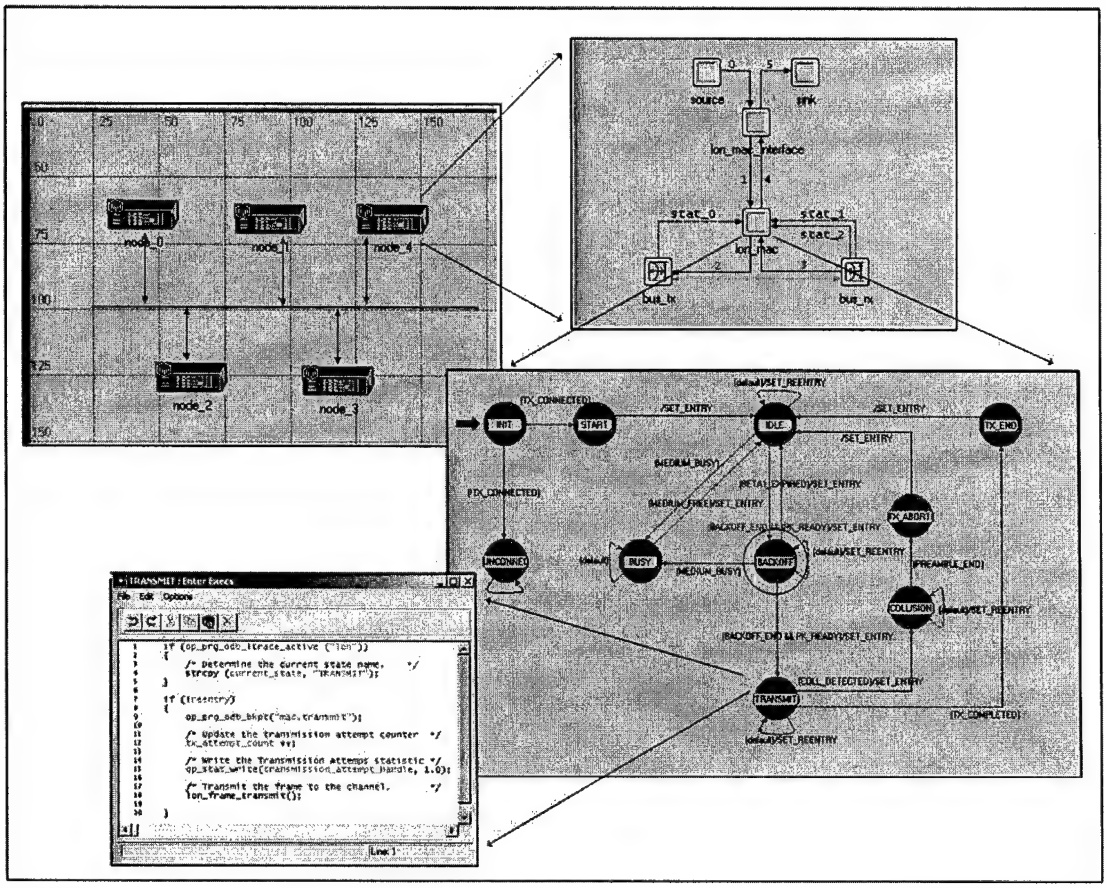


Figure 47 Hierarchical Structure of Network Model

7.2.4.4 Node Model

The node model we developed in OPNET follows the node structure we introduced in the previous section, as figure 49 illustrates. Module "bus_tx" and "bus_rx" represent a transmitter and receiver in the physical layer. Module "lon_mac" implements the p-persistent CSMA algorithm of the MAC sublayer, which will be explained in detail in section 7.2.4.4.3. Module "lon_mac_interface" is the interface module between the MAC layer and higher layers, working as a data link layer and as part of the network layer. The solid lines between different modules represent data streams, which transmit packets or frames, while dash lines notify the busy or collision status of transceivers.

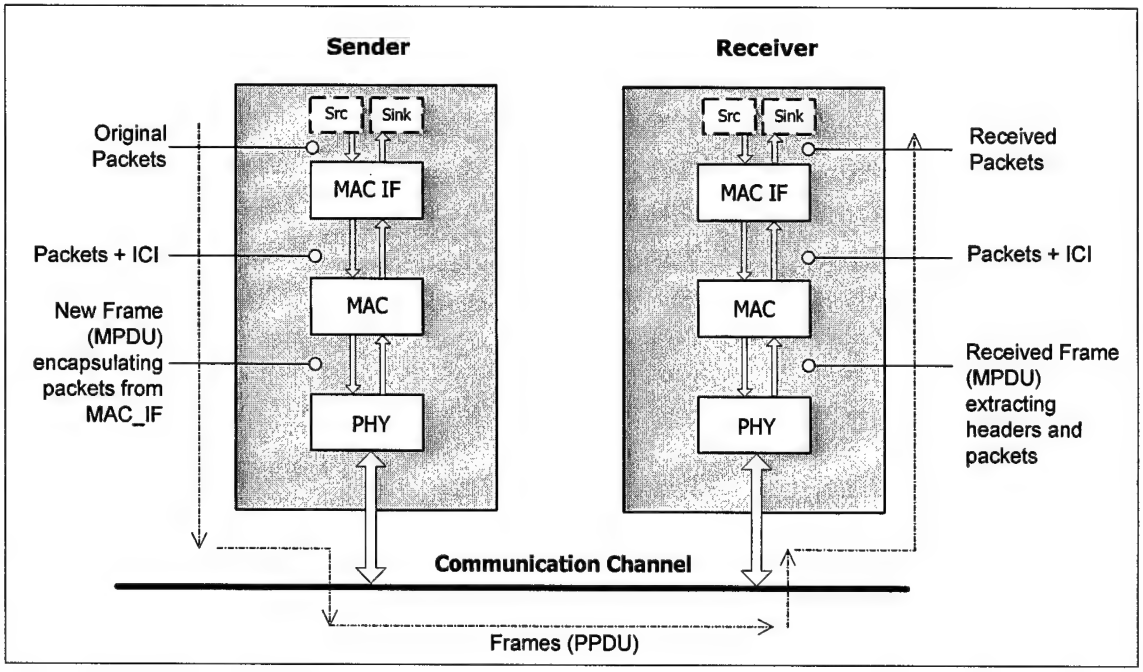


Figure 48 Data Transmission between two nodes

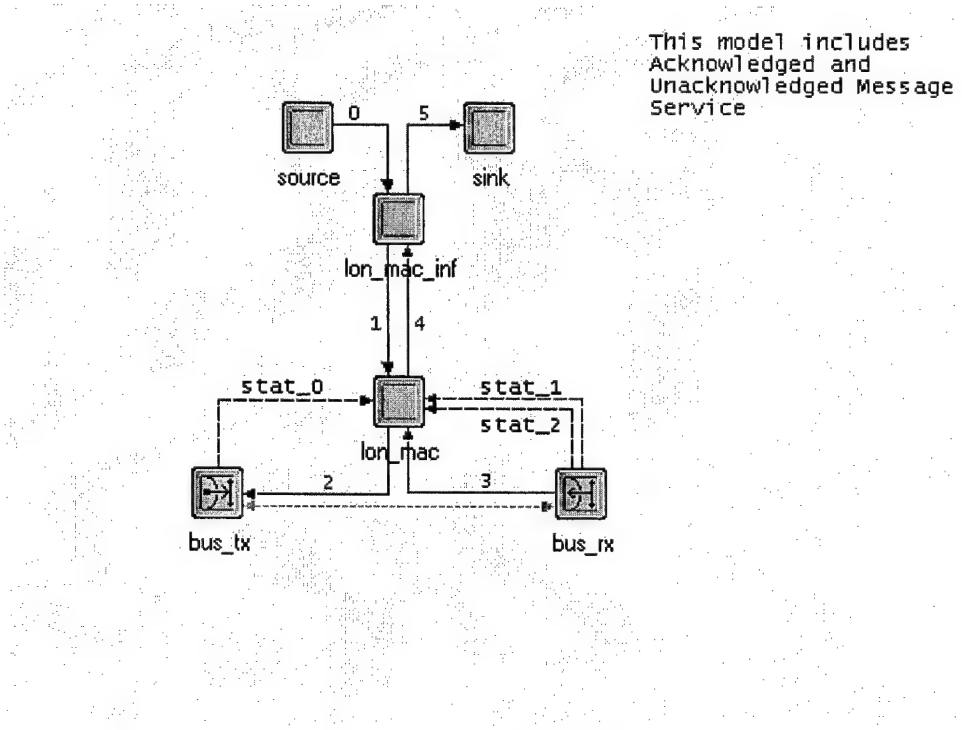


Figure 49 Layered Structure of LONWORKS device Node Model

7.2.4.4 Parameterizations

This section discusses the LONWORKS MAC related configurable parameters available as part of the simulation model suite. Note that there are two types of attributes provided by OPNET for the purpose of developing parameterized models: model attributes and simulation attributes.

Model attributes: have a scope that is local to each process model. During simulation, model attributes appear simply as attributes of queues and process modules. They may be accessed both to modify and to obtain their value. Model attributes can be declared for a process model by using the edit model attributes operation of the process editor described in Par I.

Simulation attributes: have the purpose of providing a parameterization mechanism at the system level, rather than at the level of system building blocks.

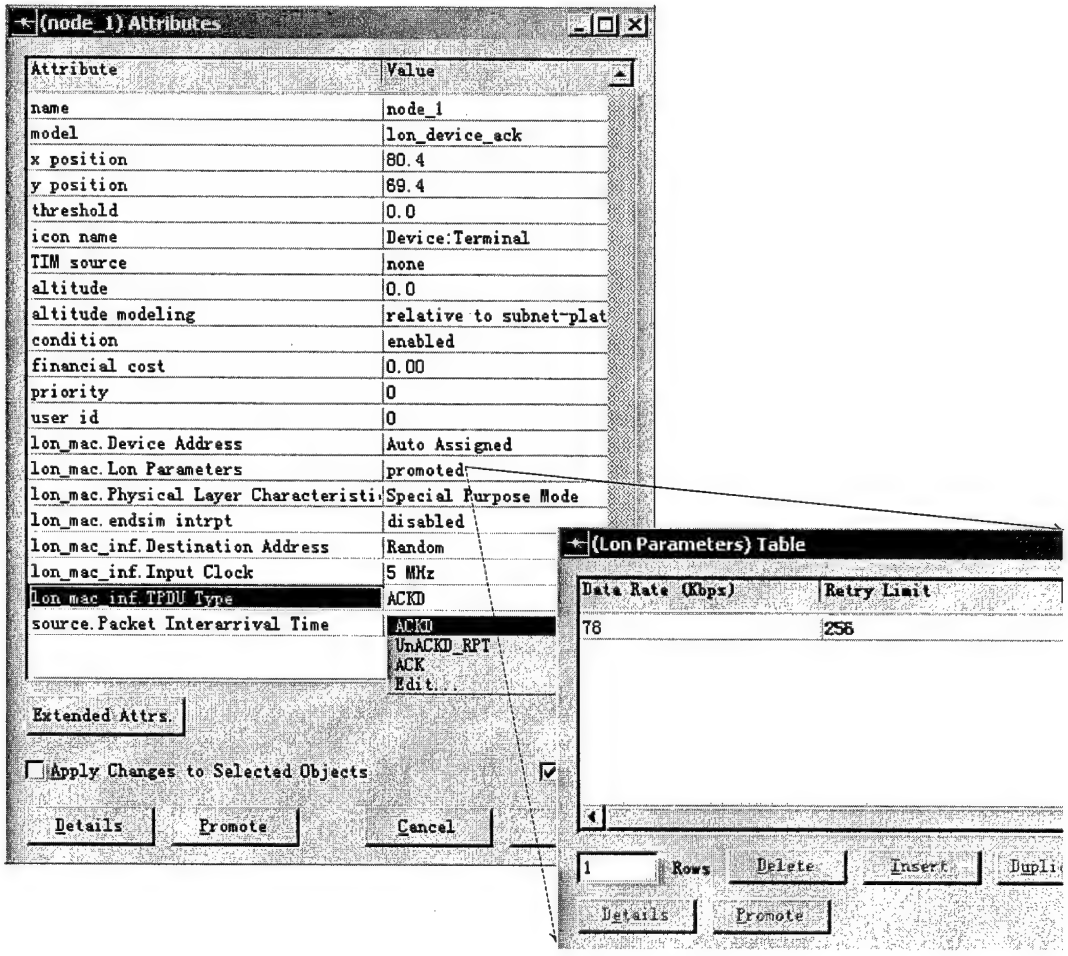


Figure 50 Various MAC attributes

Each LONWORKS node has its own set of attributes. The attributes come from different modules inside of the node model. In Figure 50, we can see a LONWORKS MAC address parameter that is an internal node address and is usually set as auto assigned unless a specific configuration is required. The destination address of each packet generated from this node can be set as random

unless a specific target is assigned. At the same time, both the optional collision detection feature and the message type feature (acknowledged or unacknowledged) can be set. The clock frequency of nodes and data rate of transceiver can be set as well. The interarrival time of packets generated from the source module can be set as a model attribute or promoted to simulation attribute. When a series of packet interarrival times are assigned during simulation, a series of channel performance indices under different network loads can be outputted to a file, and corresponding curves can be plotted and observed.

7.2.4.4 Process Models of LONWORKS Node

The following table enumerates the process models used by the LonTalk MAC model suite. Additional details on these process models are provided in subsequent sections.

Table 6 The list of Process Models

lon Mac Process models		
Name	Location	Summary
source	Higher Layer Module	Provides a simple example of upper layer interfacing with the LonTalk MAC protocol. The process generates packets to the other devices in different arrival rates, by different probability distributions.
lon_mac_inf	Higher Layer Interface Module	Provides an interface function between upper layer and MAC layer. This process generates auto-addresses of devices and packet destinations.
lon_mac	MAC Module	Performs media access control for the LONWORKS communication interface in TP/FT-10 bus configuration. The module can operate in several different data rates of channels.
bus_tx	Port Module	Implements transmitter port over a bus channel
bus_rx	Port Module	Implements receiver port over a bus channel
sink	Higher Layer Module	Provides a simple example of upper layer interfacing with the LonTalk MAC protocol. The process simply discards arriving packets.

7.2.4.4.1 Source Module

The traffic generator process module was developed based on the existing *simple_source* process model that is included with OPNET modeler. It was developed to introduce a flexible way to generate traffic into the LONWORKS communication channel. The end-user can generate not only the different types of network variables, but also the types of traffic flows according to the specified probability distribution. For example, by adjusting the parameters in *Model attributes* of the *source* process node, users can generate an integer-type standard network variable data flow with an exponential distribution. Figure 51 shows the FSM diagram for this module: After initializing all

the state variables in *init* and *init2* states, the module sets a self-interrupted timer according to the distribution of the data rate for packet generation. Every time the timer expires, it generates a pre-defined type of packet and forwards the packet to *lon_mac_inf* module.

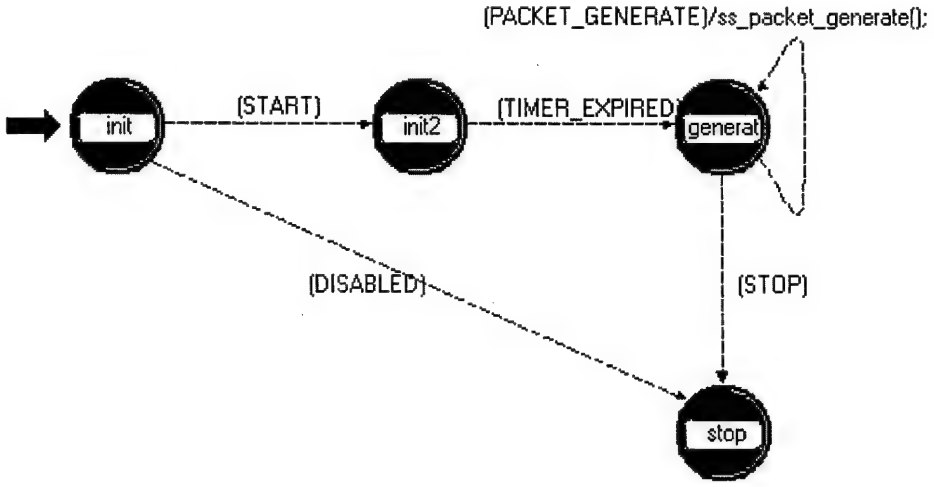


Figure 51 Process Model of *source* Module

7.2.4.4.2 lon_mac_interface Module

The FSM of the process module "lon_mac_interface" is illustrated in Figure 52. In *init* and *init2* states, the module initializes its state variables used in the entire process. Then the node registers itself and makes connections with other process nodes that are connected to the lon_mac_interface node. After leaving the initial states, the module switches among *idle*, ack_timer, and two packet processing states (*upper_arrival* state and *mac_arrival* state). The module enters the *idle* state and waits for an incoming event. The event can be either an incoming packet from the higher layer, an incoming packet from the MAC layer, or the expiration of the clock timer when using acknowledged message service. When a packet arrives from the upper process, the UPPER_LAYER_PKT_ARVL event is triggered, and the state machine enters the *upper_arrival* state where the type of packet is determined and the required processing and encapsulation are executed. Then the packet is forwarded to the MAC layer where it will be queued and will await a transmission opportunity.

During the operation, if unacknowledged message service is selected, the module only handles packets from two directions and switches the states among *idle*, *upper_arrival* and *mac_arrival*. If acknowledged messages are being used, every time the module receives a packet from the source, it will add an entry to a model maintained FIFO queue according to the destination address of this packet. At the same time, the process will start a virtual clock timer based on self-interrupts to wait for an acknowledgement from the receiving node. When an acknowledgement-type packet is received before this timer expires, MAC_LAYER_PKT_ARVL event is triggered and the corresponding entry for that ACKD-type packet will be erased from the queue. However, if TIMER_EXP is triggered, the expired ACKD-type packet will be retransmitted and a new timer will be started.

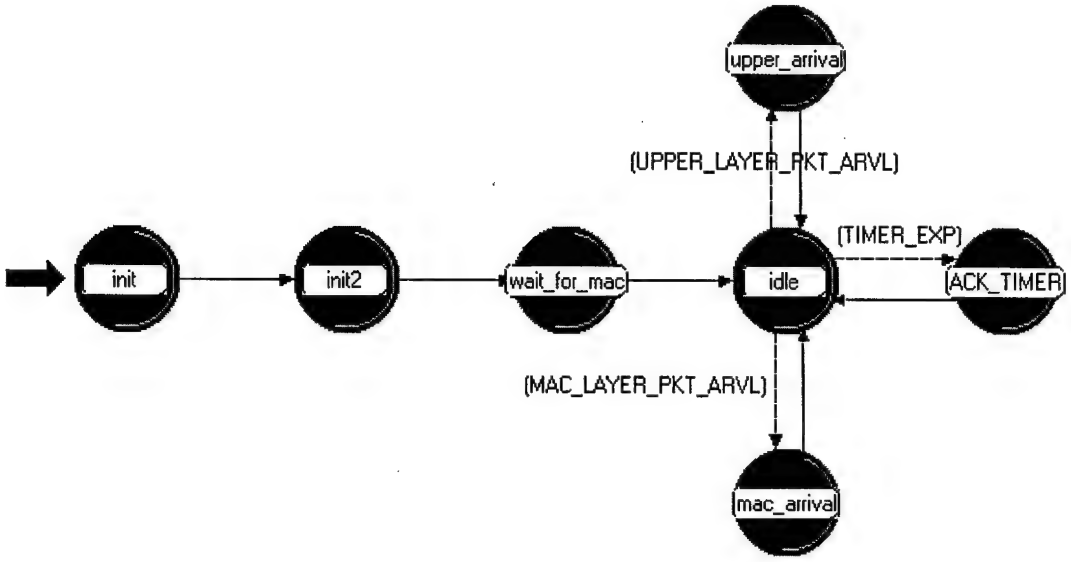


Figure 52 Process Model of *lon_mac_inf* interface Module

7.2.4.4.3 lon_mac Module

Next we discuss the operation of the finite state machine of the MAC sublayer. As we can see in Figure 53, we divide the states (other than the initialization states) into two stages. Stage 1, consists of states IDLE, BUSY and BACKOFF. The purpose of this stage is to "grasp" the channel and send a frame waiting to be sent. Entering the IDLE state means that the node is waiting a Beta1 duration to confirm that the channel is indeed idle. If the channel is still idle after a Beta1 period, the node enters the BACKOFF state to schedule a random backoff before either sending the frame or entering the BUSY state. If the node is in BUSY state, it means that another node is transmitting a frame into the channel. The node receives this frame and waits for the end of its transmission, i.e. the channel becomes idle again. While in the BACKOFF state, if the random time designated by the backoff timer expires before the channel is "grasped" by other nodes, then the node can send its frame if it has a frame waiting to be sent. If the channel has been "grasped" by another station, we reenter BUSY state to wait for another chance.

In each state, the station has the ability to monitor the channel and to process frames received from the channel. A buffer is maintained in this stage and packets from higher layers are first queued in this buffer. If the buffer is full, then packets will be dropped. This simulates the limited processing ability of a node. A backlog variable is maintained here to estimate the traffic load of the network and to produce the corresponding backlog.

The other one, stage2, consisting of states TRANSMIT, TX_END, COLLISION and TX_ABORTED, is designed to process the frame being transmitted. With the collision detection feature, a collided frame can be stopped instantly or after a preamble and arrange for retransmission. After a successful transmission, the station will enter the IDLE state and contend to transmit the next frame in the queue.

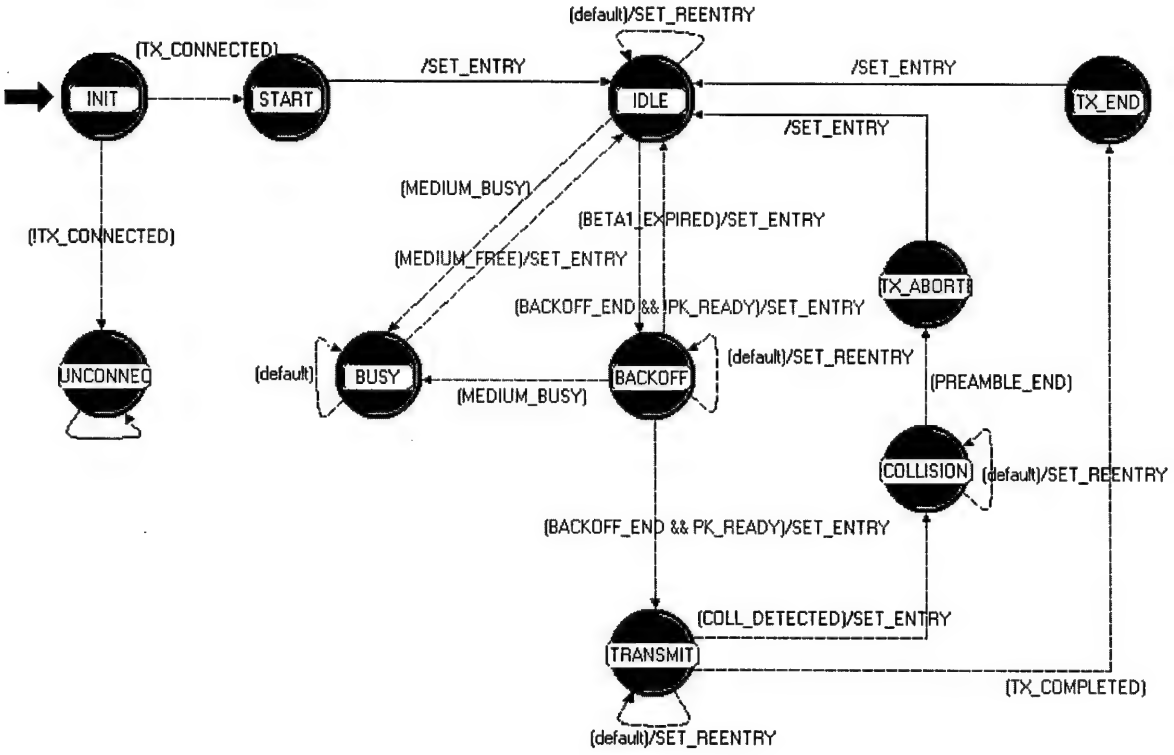


Figure 53 Process Model of *lon_mac* module

7.2.4.4.4 sink Module

The FSM of the process model *sink* module is illustrated in Figure 54. This is a simple FSM structure. All it does just accepts the incoming packets from *lon_mac_inf* module and destroys them.

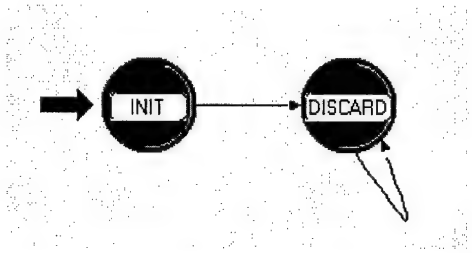


Figure 54 Process Model of *sink* Module

7.2.4.5 Process Model of Protocol Analyzer

The LonManager Protocol analyzer (LPA) in hardware implementation allows manufacturers, system integrators, and end-users to observe, analyze and diagnose the behavior of installed physical LONWORKS networks. In our modeling tool set, the simulator of LPA allows users to

perform the same functionalities as well obtain as detailed statistics, channel collision rate detection, estimation of backlog window size *etc,* which allows users to understand better how the network behaves.

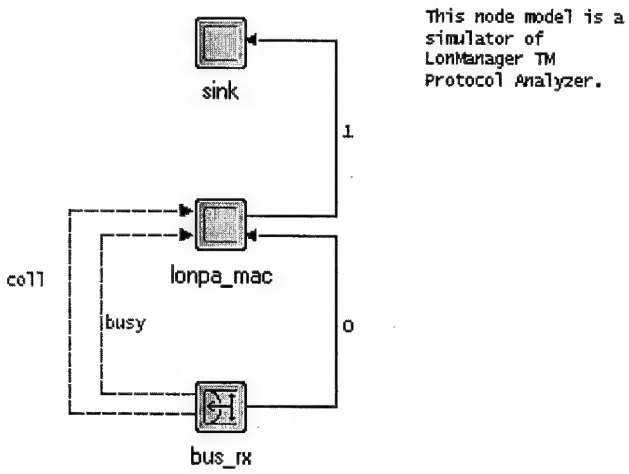


Figure 55 Layered Structure of LonManager Protocol analyzer

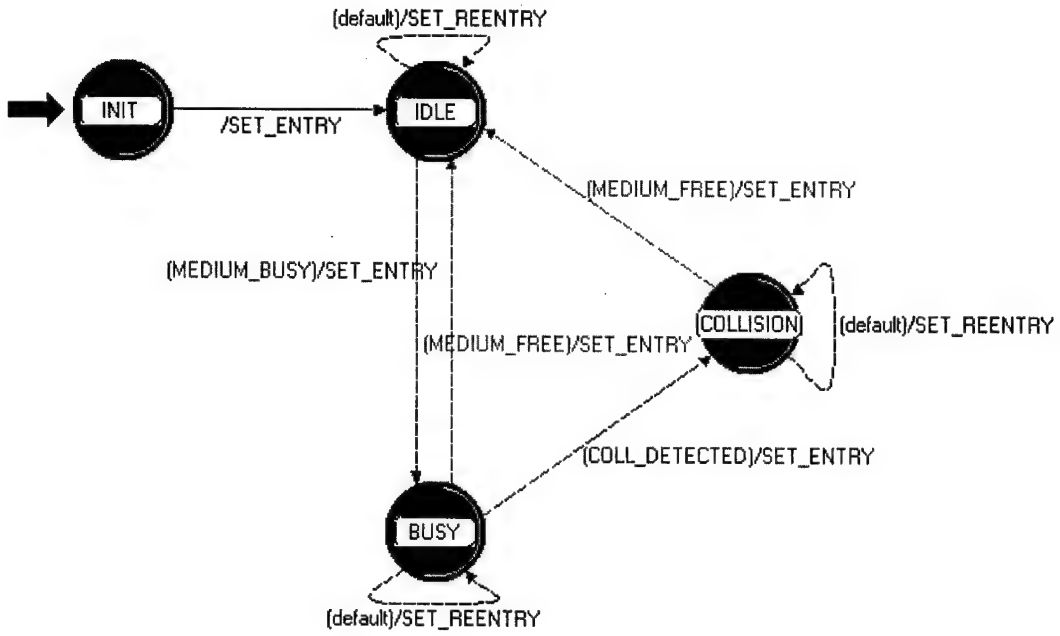


Figure 56 FSM of LPA model

In our implementation, a sophisticated transaction analysis system examines each packet as it arrives and associate related packets to aid the users or developers in understanding and interpreting traffic pattern. The simulator analyzes the communication channel by collecting, time stamping, and saving all information of network communications into either OPNET *vector* or *scalar* files

that can be viewed and analyzed. Multiple LPA simulators can be used at the same time, which allowing end-users to collect packets from multiple channels.

We show the layered structure of the node model for the LPA simulator in Figure 55. Module "bus_rx" represents the receiver in the physical layer of the LPA. Module "lonpa_mac" implements the FSM (Figure 56) for the LPA. Compared with the node model we developed for *smartcontrol* devices, the LPA node model does not come with a transmitter or any packet generators since all the protocol analyzer does is monitor the communication channel. Again, the solid lines between different modules represent data streams, which transmit packets or frames, while dash lines notify the busy or collision status of transceivers.

7.2.5 Model Interface

7.2.5.1 Packet Formats

The following Table 7 enumerates the packet formats used in the LonTalk MAC model suite.

Table 7 Packet Format

LonTalk MAC Packet Format	
Name	Description
lon_mac	Represents the LonTalk MAC packet format. This packet format allows for the encapsulation of higher layer protocol data and carries control fields such as source and destination address, backlog increment, message service type.

Figure 57 shows the frame format of the MPDU we designed in OPNET, which is a slightly different from what the protocol specifies. Some fields such as length, domain and full version of address format are neglected in this format under the assumption that we are using one subnet, one group and one domain in the modeling tool. We only implemented the fields that affect the traffic performance and characteristics. These fields include all the MPDU headers, which are priority bit (Pri), alternative path (Alt_Path), and increment of backlog (Delta_BL). The frame format of the MPDU also includes part of the NPDU header (source and destination address and TPDU_Type). Pri, Alt_Path, and Delta_BL have the following semantics:

Pri: 1-bit field specifying the priority of this MPDU: 0=Normal, 1=High.

Alt_Path: a 1-bit field specifying the channel to use. This is a provision for transceivers that have the ability to transmit on two different channels and receive on either one without the need to instruct the transceiver to explicitly receive on a specific channel. The transport layer sets this bit for the last two attempts (for acknowledged and request /response services), unless requested to specify the alternate path for every transmission. For any packet received that has the alt_path bit set and that requires an acknowledgment, response, challenge, or reply, the same alt_path bit shall be set in the corresponding acknowledgement, response, challenge, or reply.

Delta_BL: a 6-bit unsigned field (>=0); specifies channel backlog increment to be generated as a result of delivering this MPDU.

1	1	6	7	7	2	
Pri	Alt_Path	Delta_BL	Src_Address	Dest_Address	TPDU_Type	PDU/Data

Figure 57 Frame format of MAC layer Protocol data Unit

Figure 58 shows the packet structure implemented in the OPNET packet format editor.

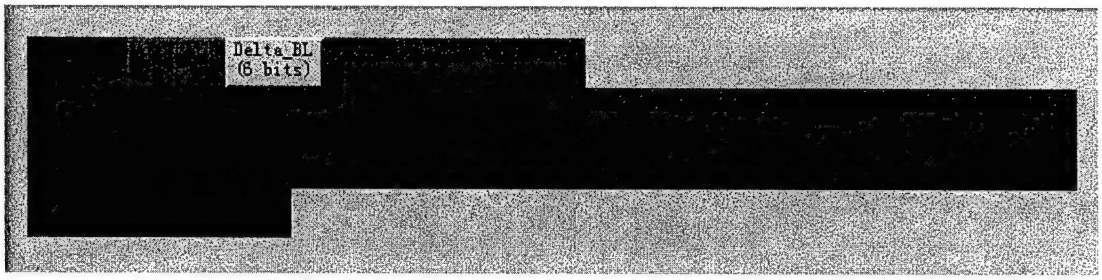


Figure 58 Frame format from OPNET

7.2.5.2 ICI Formats

OPNET uses ICI (Interface Control Information) format to formalize interrupt-based inter-process communication. Defined in the format are the names of fields within the ICI, their data types, and their default values. The following Table 8 enumerates the ICI format used in the LonTalk MAC model suite. Figure 59 shows the ICI format we designed in the OPNET ICI editor for lon_mac_request. Figure 60 shows the ICI format for lon_mac_ind.

Table 8 ICI Fomrat

LonTalk MAC ICI Format	
Name	Description
lon_mac_request	Used in conjunction with packet transfer from ***lon_mac_inf*** to the ***lon_mac.*** The attributes carried in this ICI specify the source and destination address, type of upper layer protocol data packet.
lon_mac_ind	Used in conjunction with packet transfer from ***lon_mac*** to the ***lon_mac_inf.*** The attributes carried in this ICI specify the information along with the packet received by MAC layer

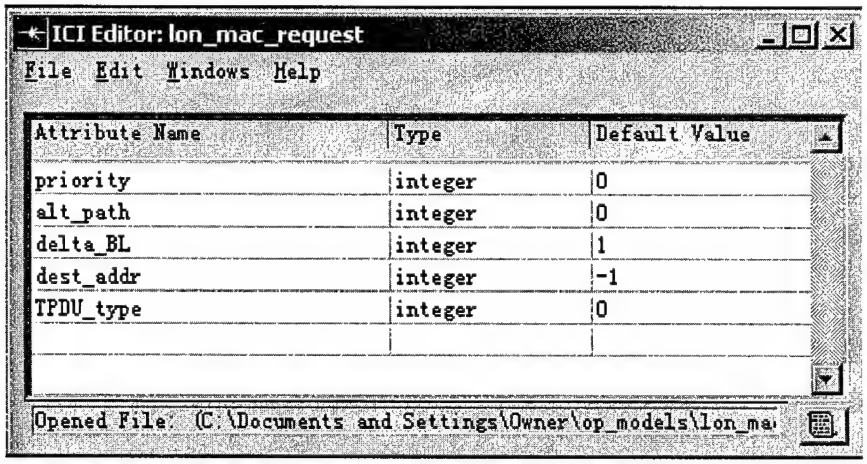


Figure 59 ICI Attributes of lon_mac_request

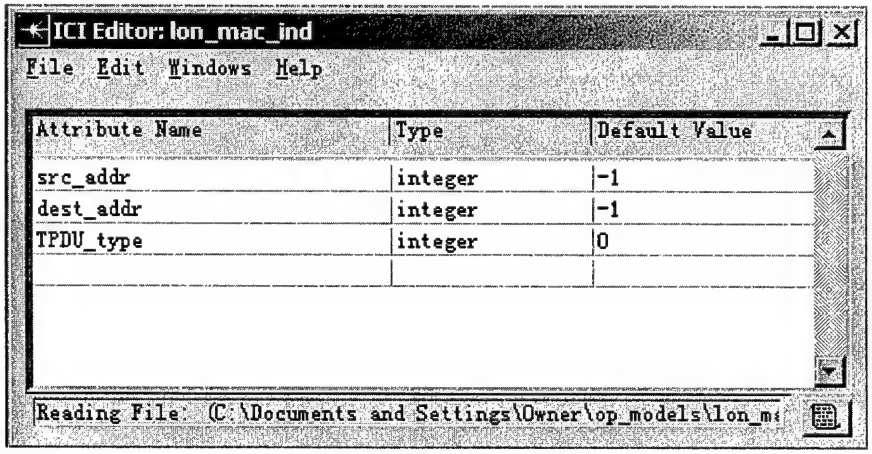


Figure 60 ICI Attributes of lon_mac_inf

7.2.6 Available Statistics

The usefulness of a simulation model is dependent on the statistics it provides of the physical system. This is connected to the statistics it is able to generate when it runs. In our LONWORKS modeling tool we have implemented a comprehensive statistical set of "points of measurements". These statistics are stored in two types of output files: *vector* and *scalar*. *Vector* output files trace the course of a statistic over an interval of time; each data point has an associated time at which it was logged. *Scalar* output files store a set of singular statistic values grouped by simulation. Table 9 lists the statistics being outputted to the vector files, while Table 10 lists the statistics being outputted to the scalar files. Figure 61 shows the OPNET implementation of available statistics.

Table 9 Statistics stored in vector output file

Name	Summary
Backlog Size	Index of backlog window size that the current node can randomly choose from before transmission while contending for the medium.
Collision Count	Number of collisions encountered by the mac layer of this node.
End-to-End Delay (sec)	End-to-End delay of the packets accepted by this node.
Packets Offered per Node (packets)	Total number of packets sent by current node.
Packets Offered per Node (packets/sec)	Average number of packets sent by current node.
Packets Received per Node (packets)	Total number of packets received by current node.
Packet Received per Node (packets/sec)	Average number of packets received by current node.
Queue Size of Packets being held	Number of packets received from higher level being held at queue of MAC layer.
Traffic Offered per Node (bits)	Total data traffic (in bits) sent to the mac layer from a higher layer.
Traffic Offered per Node (bits/sec)	Averagedata traffic (in bits) sent to the mac layer from a higher layer.
Traffic Received per Node (bits)	Total number of bits forwarded to higher layer by the mac layer.
Traffic Received per Node (bits/sec)	Average bits per second forwarded to the next-higher layer by the mac layer in this node.
Transmission Attempts	Number of transmission attempts made by the mac layer of current node before frames are successfully transmitted.

Table 10 Statistics stored in scalar output file for LONWORKS node device

Name	Summary
Average Offered Load (packets/s)	Average packet rate offered by the source as channel load
Collision Rate (%)	Collision rate of this specific node, percentage of collisions with ratio to the total times of transmission attempts
Node Throughput (packets/s)	Throughput of this specific node
Channel Throughput (packets/s)	Total throughput measured on the channel
Dropped Packets (%)	Percentage of packets that are dropped due to fullness of buffer

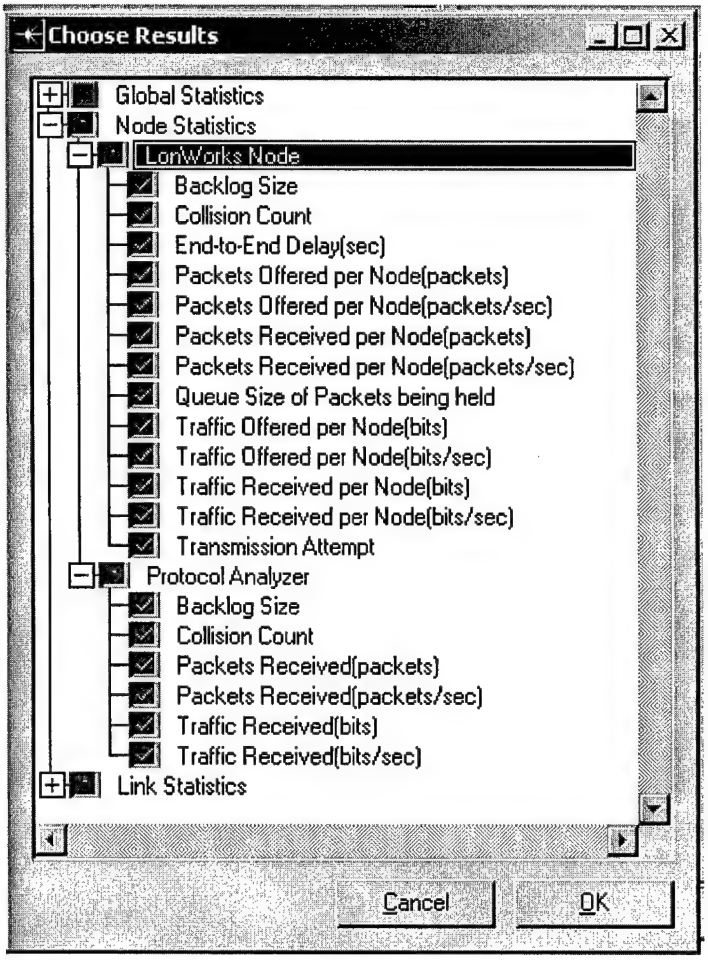


Figure 61 Collectable Statistics in LONWORKS Modeling Tool

7.2.7 Files Included in the Model

This section gives a brief overview of the files required for this model suite in **<opnet_dir>/op_models/** and **<opnet_dir>/op_models/lon_mac_ack** directory.

lon_link_tp.lk.m	Bus channel and bus link tap model file
lon_mac.pk.m	Model file for MAC packet format
lon_mac_ind.ic.m	ICI format file of lon_mac_ind
lon_mac_request.ic.m	ICI format file of lon_mac_request
lon_device.nd.m	Node model file
lon_mac.pr.m lon_mac.pr.c lon_mac.i0.pr.o	Model file, source code and compiled object file for MAC process model

lon_mac_inf.pr.m lon_mac_inf.pr.c lon_mac_inf.i0.pr.o	Model file, source code and compiled object file for MAC Interface process model
lon_mac_<net_name>.prj	Project model file
lon_mac_<net_name>-<scenario>.nt.m	Network model file for a specific scenario
lon_mac_<net_name>-<scenario>.seq	Simulation sequence file
<simulation_output>.ov	Output vector file generated by simulation in order to store time-series statistical data
<simulation_output>.os	Output scalar file generated by simulations in order to store individual values that capture summarized behavior, performance, or parameterization of a simulation

7.3 Analytical Models of LONWORKS p-CSMA algorithm

To validate the performance of OPNET-based LONWORKS modeling tool, two analytical models of the simplified version of LONWORKS protocol were developed. The first analytical model is developed from the viewpoint of the communication channel, from which we derived the probability of collision on the channel; the second analytical model is developed from the viewpoint of the node that is connected to the channel. The analytical approach that we followed here is based on [12].

7.3.1 The Behavior of communication channel

In LonTalk's CRA, each packet-ready node that is connected to the channel detects whether or not the common channel is idle, and then waits for $s\beta_2$ seconds before it attempts to transmits its message. Here s is a random number, selected from $\{1, 2, \ldots, S\}$, and β_2 is a fixed constant. When the nodes use an unacknowledged message service, the backlog window size S of each node remains in the lowest level most of the time during the transmission and is likely equal to 16 (this number may be adjusted due to the collisions).

When the node with the smallest value of s gains access to the channel (provided it is the only node that selected this value of s) it will transmit. The other nodes will sense the active transmission, begin receiving the message, and temporarily abort their attempt to transmit. Nodes assume that a transmission is complete when the channel has been idle for β_1 seconds (β_1 is a fixed constant). Once β_1 seconds have elapsed, each node will select a new value of s, and again wait for $s\beta_2$ seconds before attempting transmission. If two nodes selected the same (lowest-value) randomization slot s—and thus attempt to transmit at the same time—a collision occurs. When a collision occurs, all nodes wait for the channel to remain idle for β_1 seconds before restarting a new random $s\beta_2$ waiting period. If we use the following notation:

- N = the number of nodes in the network
- s = an index; time since the channel was idle $s \in \{1, 2, \ldots, S\}$
- S = the number of slots in the randomization set (S=16),

Assuming the network is operating under a heavy steady state load and N nodes are competing for the channel, it can be shown that for a fixed S, the probability that a collision has occurred in the s^{th} time slot after the channel returned to the *idle* state is:

$$P_{ch\_coll} = \sum_{s=1}^{S} \sum_{n=2}^{N} P_{ns} \quad , \qquad \text{(Eq-1)}$$

where P_{ns} is the probability that $n>1$ nodes collide in the s^{th} slot, expressed by the following equations:

$$P_{ns} = \begin{cases} \left(\frac{1}{S}\right)^n, & \text{for } s = S \text{ and } n = N \\ \frac{N!}{n!(N-n)!}\left(\frac{1}{S}\right)^n \left(\frac{S-s}{S}\right)^{N-n}, & \text{otherwise} \end{cases} \qquad \text{(Eq-2)}$$

7.3.2 The Behavior of single node connected to channel

When the nodes connected to the channel use message type service other than unacknowledged, the backlog window size of each node will not maintain the lowest level. The number of slots in the randomization set S varies according to the different values of backlog window size. A node-oriented model needs to be developed in order to analyze the change of window backlog size. The key assumption used in this analytical model is that, for a given node that operates under LONWORKS protocol, a transmission always collides with probability p regardless of the backlog window size used to choose the random waiting period for the reservation attempt. We divide our analysis of the LONWORKS communication channel into two parts. In section 7.3.2 we consider the behavior of a single node that is connected to a LonTalk bus channel to compute the collision probability p and the stationary probability τ that the station transmits a packet in a randomly chosen time. This probability is independent of the reserved access scheme employed by the stations. Then, by examining the events that can occur in a randomly chosen backlog window, throughput performance is expressed as a function of probability τ.

We concentrate on "saturation throughput" for a fixed number of stations n. This is a fundamental performance index defined as the limit reached by the system performance throughput as the offered load increases, and represents the maximum load that the system can carry in stable conditions. In saturation, every node that is connected to the bus channel always has a packet available for transmission immediately after the completion of each successful transmission.

Let $b(t)$ be the stochastic process that represents the backlog time counter for a specific station. Process $b(t)$ represents the remaining backlog time before the node starts to transmit the packet. A discrete, integer time scale is adopted where t and $t+1$ correspond to two consecutive slot times, and the backlog time counter of each station decreases at the beginning of each slot time. The

backlog window size for every station depends on the collisions and on the successful reservation attempts experienced by the station in the past. As a result, process *b(t)* is non-Markovian. Following the control-networking standard of ANSI/EIA 709.11, we define for convenience $W_i = W_{base}(1+i)$, where W is the base window size of the backlog counter and, by default, is equal to 16. The maximum backlog value '*m*' is 63. We designate *s(t)* to be the stochastic process representing the backlog stage (0,..., *m*) of the station at time *t*.

As we mentioned before, the key assumption of this model is that a packet transmission collides with the same probability *p* regardless of the backlog time counter used for this transmission. Based on this assumption, we modified Bianchi's discrete time Markov chain model [14], which was originally developed to study the MAC algorithm of IEEE 802.11 protocol for wireless local area networks. we can states {*s(t), b(t)*} by this discrete time Markov chain model presented in Figure 62.

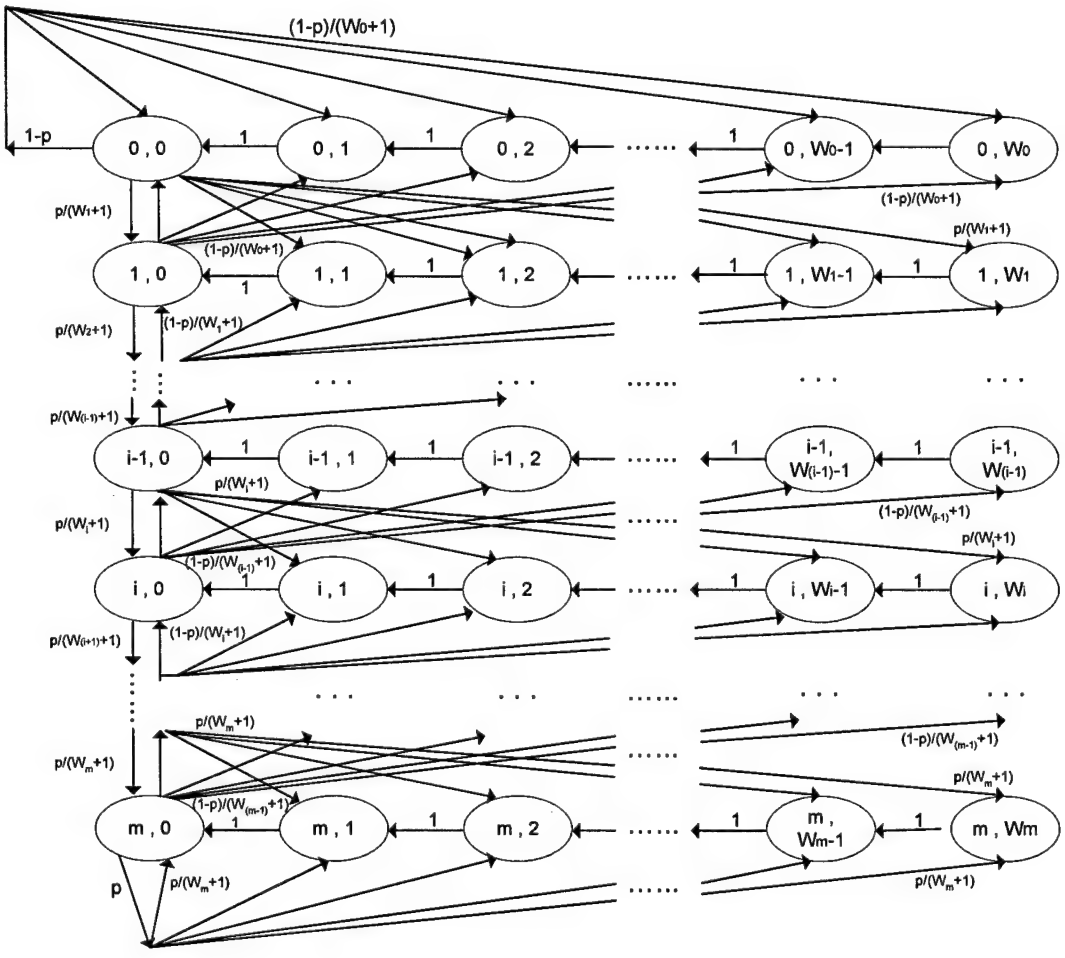


Figure 62 Markov Chain model for the backlog window size

Figure 62 shows that when a node connected to the channel wants to transmit a packet, under the conditions of saturation, it will generate a random waiting period (rwp) according to the backlog window estimation. It then decreases the rwp from the value of the backlog window size for the entire slot unit until it reaches zero. After a successful transmission attempt, the backlog stage is decreased and the new rwp value is uniformly chosen from the range size decreased by W_{base}. The model also shows that after an unsuccessful transmission attempt, the new rwp will be chosen from a larger range. After a successful transmission attempt at stage 0, the new rwp will be chosen from a same range and after an unsuccessful transmission attempt at stage m the new rwp will not be further increased.

If we designate the short notation $P\{i_i, j_i \mid i_0, j_0\} = P\{s(t+1) = i_i, b(t+1) = j_i \mid s(t) = i_0, b(t) = j_0\}$, we can find the transition probabilities from Figure 62, which are:

$$P\{i, j \mid i, j+1\} = 1,\ i \in (0, m), j \in (0, W_i - 1) \quad \text{(Eq-3)}$$

$$P\{i, j \mid i+1, 0\} = \frac{1-p}{W_i + 1},\ i \in (0, m-1), j \in (0, W_i), \quad \text{(Eq-4)}$$

$$P\{i+1, j \mid i, 0\} = \frac{p}{W_{i+1} + 1},\ i \in (0, m-1), j \in (0, W_{i+1}), \quad \text{(Eq-5)}$$

$$P\{0, k \mid 0, 0\} = \frac{1-p}{W_0 + 1},\ j \in (0, W_0), \quad \text{(Eq-6)}$$

$$P\{m, j \mid m, 0\} = \frac{p}{W_m + 1},\ j \in (0, W_m), \quad \text{(Eq-7)}$$

Equation-1 states that after the node generates the random waiting timer according to the estimated backlog window size, it keeps decreasing until it reaches the zero state of the current layer. If the transmission of the packet at time i is successful (probability is 1-p), the node moves one stage toward stage 0 and the estimated backlog window size (represented by b(t)) is decreased by 1. The new random waiting period for the transmission of the next packet will be generated uniformly from a smaller range (0, W_i), which is represented in Equation-2. Equation-3 shows that if the transmission of the packet has failed, which means that a collision has occurred with probability p, the node moves to a lower stage and the estimated backlog window size is increased by 1. The new random waiting timer will be generated from a larger range (0, W_{i+1}). Equation-4 considers the special condition of the node being at stage 0 when it transmits a packet successfully. Equation-5 is the special case when the node is at the last stage and it fails to transmit a packet.

We designate $b_{i,j} = \lim_{t \to \infty} P\{s(t) = i, b(t) = j\}$, $i \in (0, m)$, $j \in (0, W_i)$ to be the stationary behavior after the process has been running for a long time. After some algebra and applying the method of induction, it can be shown that

$$b_{i,0} = \left(\frac{p}{1-p}\right)^i b_{0,0}, . \quad \text{(Eq-8)}$$

From backlog stage s(t), t=0, we can establish another equation for steady state probability

$$b_{0,j} = \frac{W_0 + 1 - j}{W_0 + 1}[b_{0,0}(1-p) + b_{1,0}(1-p)] \qquad \text{(Eq-9)}$$

From backlog stage s(t), t=m, it satisfies

$$b_{m,j} = \frac{W_m + 1 - j}{W_m + 1}[b_{m,0}p + b_{m-1,0}p] \qquad \text{(Eq-10)}$$

For the remaining of s(t) ($t \in (1, m-1)$), we have

$$b_{i,j} = \frac{W_i + 1 - j}{W_i + 1}[b_{i-1,0}p + b_{i+1,0}(1-p)]. \qquad \text{(Eq-11)}$$

Substituting Equation-6 into the above three equations, we get one simplified form:

$$b_{i,j} = \frac{W_i + 1 - j}{W_i + 1} b_{i,0},\ i \in (0, m), j \in (0, W_i), \qquad \text{(Eq-12)}$$

Since the sum of the probabilities of all the states in Markov Chain model is equal to 1, we have,

$$\sum_{i=0}^{m} \sum_{j=0}^{W_i} b_{i,k} = 1, \qquad \text{(Eq-13)}$$

From which we can further derive

$$\frac{b_{00}}{2} \sum_{i=0}^{m} \left(\frac{p}{1-p} \right)^i (W_i + 2) = 1. \qquad \text{(Eq-14)}$$

Since

$$W_i = BL(1+i) - 1,\ i \in (0, m), \qquad \text{(Eq-15)}$$

we have

$$b_{00} = \frac{2}{(BL+1)\sum_{i=0}^{m} \left(\frac{p}{1-p} \right)^i + BL \sum_{i=0}^{m} i \left(\frac{p}{1-p} \right)^i} \qquad \text{(Eq-16)}$$

We can now express the probability τ that a station transmits in a randomly chosen time slot. As any transmission occurs when the backlog time counter is equal to zero, regardless of the backlog stage, it is:

$$\tau = \sum_{i=0}^{m} b_{i,0} \qquad \text{(Eq-17)}$$

After substituting and applying the equation of Arithmetic-Geometric series, τ can be expressed as

$$\tau = \frac{2}{(BL+1)+\frac{BLp((1-p)^{m+1}+(2m+1)p^{m+1}-(m+1)p^{m})}{((1-p)^{m+1}-p^{m+1})(1-2p)}}. \qquad \text{(Eq-18)}$$

Probability τ depends on collision probability p which is still unknown. The probability p that a reservation attempt collides is the probability that at least one of the remaining *n-1* stations transmit in the same time slot. Hence, we are able to get another equation for p and τ:

$$p = 1-(1-\tau)^{n-1} \qquad \text{(Eq-19)}$$

Equations 10 and 11 form a nonlinear system of the unknown p and τ. The system can be solved by employing numerical methods evaluating p and τ. It is easy to prove that there is a unique solution to this nonlinear system since, in Eq-10, τ is a continuous and monotonically-increasing function in the range of $p \in (0,1)$. In Equation-11, τ is a continuous and monotonically-decreasing function in the same range.

7.3.3 Throughput analysis of the LonTalk bus channel

Let S be the normalized system throughput.
Based on the p and τ evaluated in Section 2.1, throughput efficiency can be evaluated. We define P_{tr} as the probability that at least one transmission attempt occurs in the given time slot. For a LONWORKS network of n nodes, each transmitting with probability τ, P_{tr} is given by

$$p_{tr} = 1-(1-\tau)^{n}. \qquad \text{(Eq-20)}$$

The probability P_s, that an occurring transmission is successful, is given by the probability that one node transmits and the remaining n-1 nodes remain silent provided that at least one transmission occurs in the channel:

$$P_s = \frac{n\tau(1-\tau)^{n-1}}{P_{tr}}. \qquad \text{(Eq-21)}$$

A successful transmission in a randomly selected slot occurs with probability $P_{tr}P_s$ and the time to transmit a packet is given by $P_{tr}P_s(pkt\_length/c)$, where *pkt_length* is the length of a packet and c is the data transfer bit-rate of the LONWORKS bus channel (for TP/TF-10 channel, the bit-rate is 78*kbps*) . The average slot duration can be evaluated by considering that with probability $1-P_{tr}$ the slot is empty, with probability $P_{tr}P_s$ the slot contains a successful transmission and with probability $P_{tr}(1-P_s)$ the slot contains a collision. Thus, throughput efficiency S can be evaluated by dividing the time to transmit a packet by the average slot duration

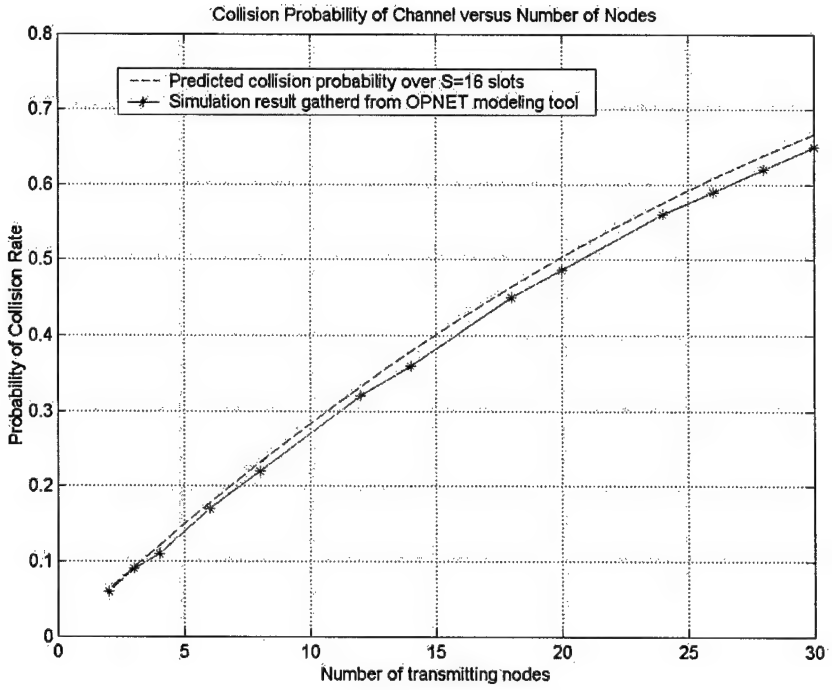


Figure 63 Collision Probability of Channel versus. Number of Nodes

Figure 64 shows the collision probability of specified node under the conditions of saturation. The dashed line represents the analytical results gathered from Equationi-18, 19. The solid line represents the result gathered from simulation. During the simulation, every node in the network always tried to transmit a packet as long as the node senses that the channel was idle. We set the parameters of simulation according to the assumption of our node-oriental analytical model; The service type of transmission was "acknowledged", which results in dynamic adjustment the size of randomizing window to the current estimated channel backlog BL ranging between 1 and 63. The simulated nodes on the channel were set to 5MHz again and TP/FT-10 media-type network was still used.

From the figure we can see that the simulation results match the trend of the analysis, with greater discrepancy for networks with small number nodes connected on it. In these small networks, the underlying assumption of our analysis—namely that every node is ready to transmit whenever the channel is available—does not hold due to hardware limitations[4]. As a result actual collision rates are lower than those predicted by Equation-18 and 19. For networks with large number of nodes, the channel is occupied with activity and now the channel, not the local node processors, becomes the limiting factor in the system. Messages in the media access processor begin to queue up, guaranteeing that whenever the channel returned to *idle*, every node in the network would position itself in the transmission queue.

[4] Under our simulation conditions, the channel has a capacity to carry approximately 225 messages per second, but each node can only transmit approximately 60.5 messages per second. (These statistics were also obtained in experiments with physical hardware.) Under these conditions, nodes often do not compete for the channel, thereby lowering the probability of collision. The node processor responsible for physical media access is often idle as it waits for the other processors in the node to prepare the next message.

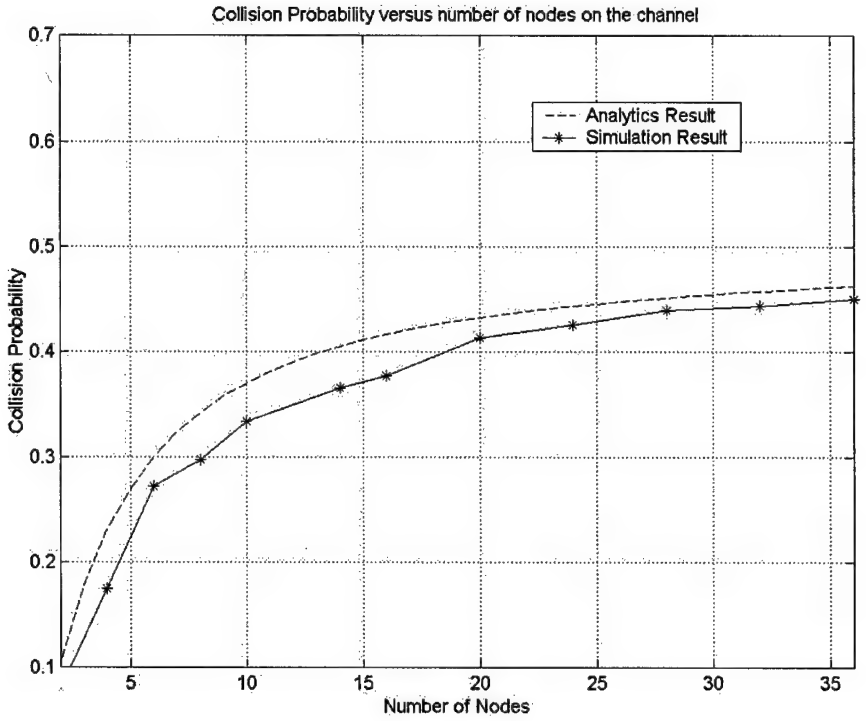


Figure 64 Collision Probability of Node versus. Number of Nodes

7.4.2 Validation of Channel Throughput and Bandwidth Utilization

In order to compare the accuracy of our modeling tool, we compared the results of simulation to the results gathered from measurement of physical system. We demonstrate two examples in this section. One uses six (6) programmable *smartcontrol* devices while the other uses eight (8) devices. In both cases, all the *smartcontrol* devices were connected with twisted-pair wiring and configured to transmit at 78.125 kbps. All the nodes are programmed to transmit floating-point type network variable to a neighboring node according to the variable rates. We configure exactly the same settings in the LONWORKS modeling tool.

Figure 65 and 66 are the result of simulation and hardware test for a bus channel with six (6) nodes connected on it plotted in one graph, which *channel throughput* (Figure 65) and *bandwidth utilization* (Figure 66) as functions of the traffic offered by each node are compared. The channel throughput is expressed in messages transmitted per second, which is the actual message load to the network. The bandwidth utilization is defined as the time used for transmission divided by the total elapsed time of simulation. In both of the plotting, the line crossed by the circle represents data gathered experimentally; the line crossed by the star represents the corresponding throughput rates predicted by our modeling tool.

Figure 67 shows the channel throughput for the case of eight (8) nodes and Figure 68 shows its bandwidth utilization.

The simulation model matches the real system well. We note that under light traffic load, the channel throughput increases linearly until a saturation point.

$$S = \frac{P_{tr} P_s (pkt\_length / C)}{(1 - P_{tr})\sigma + P_{tr} P_S T_s + P_{tr}(1 - P_s) T_c} \qquad \text{(Eq-22)}$$

where T_s is the slot duration when a successful transmission occurs, T_c is the slot duration for a collision involving two or more simultaneous frame transmissions and σ is the unit slot time of backlog window.

7.4 Model Validation

We discuss in this section numerical results concerning the performance of the LONWORKS modeling tool. In addition to the analysis presented in section 7.3.3, simulation results of our communication simulator have been compared to the results gathered from physical test platform. Verification and validation of the modeling tool are essential if the results of those simulations are to be trusted. The purpose of the comparison are twofold: 1) to cross-validate the results obtained from two analytical models, and 2) to experiment with variations of the scheme, traffic patterns, and network loads which are not easily represented by the present analysis.

7.4.1 Validation of Collision Probability

Figure 63 illustrates the collision probability of communication channel as a function of the number of nodes in the channel for the number of slots in randomization set is fixed value and equals to 16 slots. Dashed line is the analytical result gathered from Equation-1, and Solid line with '*' symbols indicates the result gathered from simulation tool. In each simulation run, every node in the network attempted to transmit repeatedly an unacknowledged floating-point network variable to a neighboring node as fast as possible. The simulated nodes on the channel were set to "5MHz neuron processors" and they are all connected to TP/FT-10 media-type network at 78.125 kbps data rate.

As expected, the collision probability increases with the number of nodes increases. The OPNET simulation results match the collision rates predicted by Equation-1,2 closely. As we can see from the figure, when the network size and number of collisions increase, the difference between the analytical result and simulation result also increases, since our assumption of a fixed S is no longer valid when there are more nodes that compete for the channel. This situation results in increasing number of collisions and backlog window size. Actually, the increase in the value of S by the protocol allows the physical network to have a better collision rate than the rate our analytical model predicted.

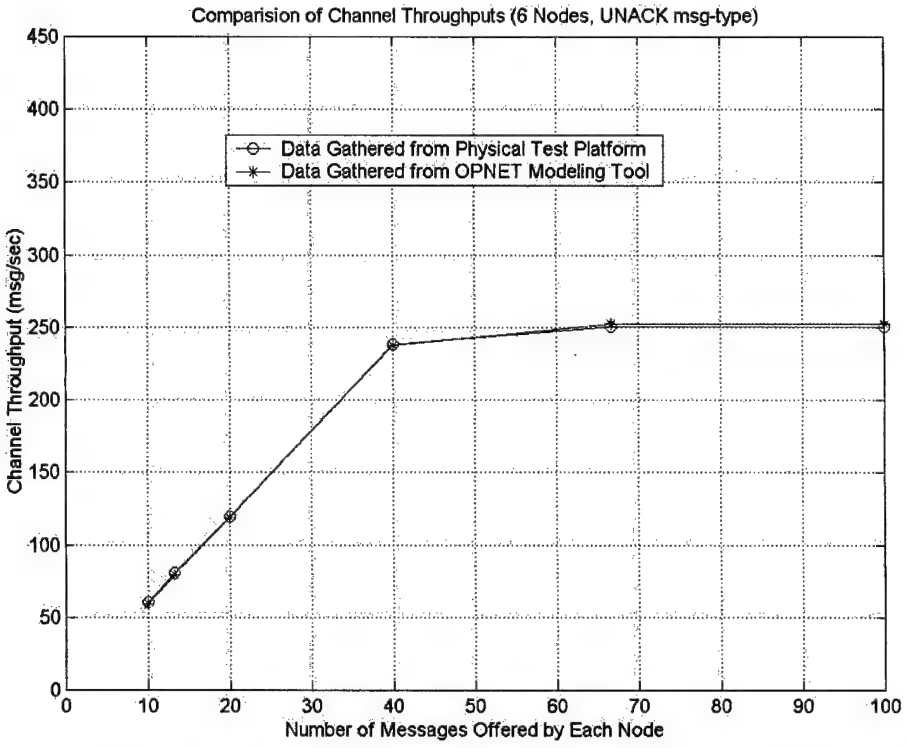


Figure 65 Channel Throughputs versus. Traffic offered by Each Node (6 Nodes, UNACK)

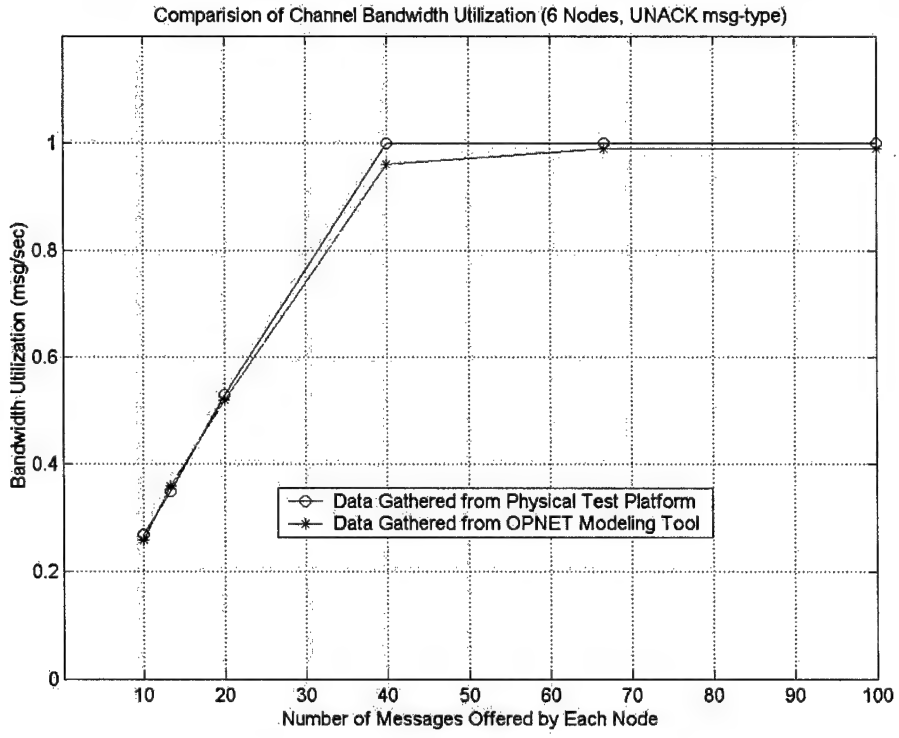


Figure 66 Bandwidth Utilization versus. Traffic Offered by Each Node (6 Nodes, UNACK)

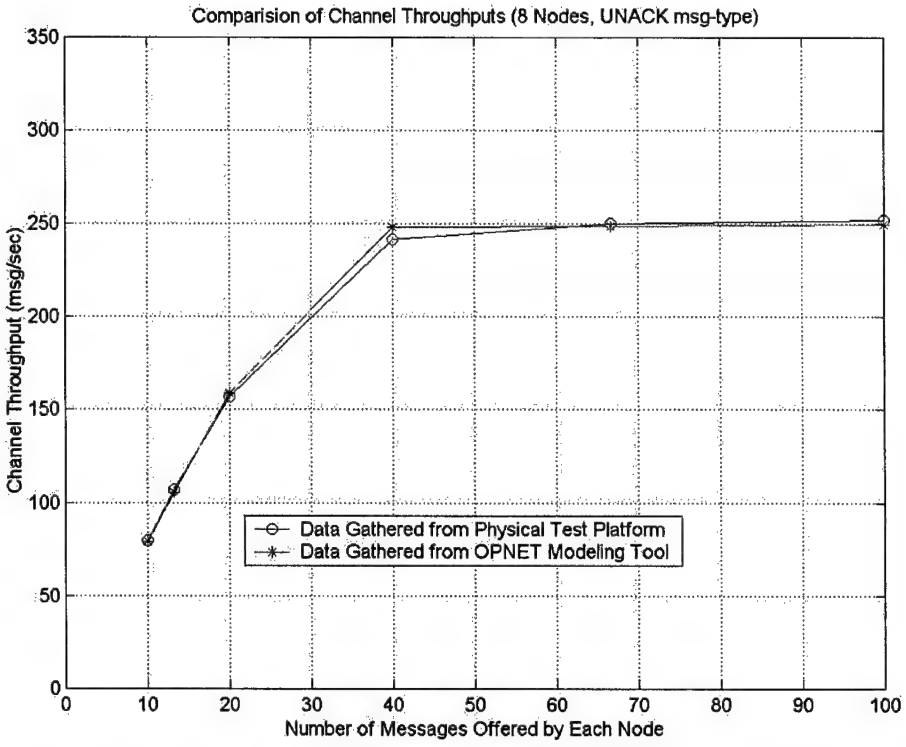


Figure 67 Channel Throughputs versus. Traffic offered by Each Node (8 Nodes, UNACK)

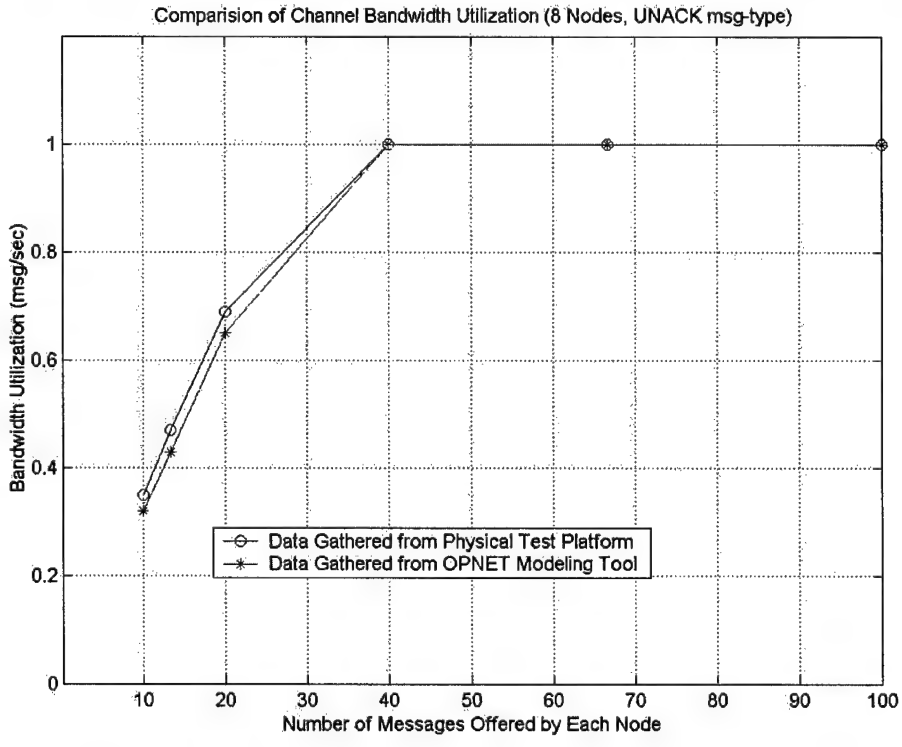


Figure 68 Bandwidth Utilization versus. Traffic Offered by Each Node (6 Nodes, UNACK)

7.4.3 The Case Study

In this section, we present a case study of the communication simulator. We studied a bus topology channel and four nodes connected on it.

During the experiment, we assume that each node transfers standard floating temperature-type data to its neighboring node at the data rate of 60 messages/sec. All the nodes use the UNACK service to communicate with each other.

After the network by using the GUI of the communication simulator (Figure 46), we initialed the parameter settings in the window of node attributes (Figure 50), and ran the simulation 60 seconds.

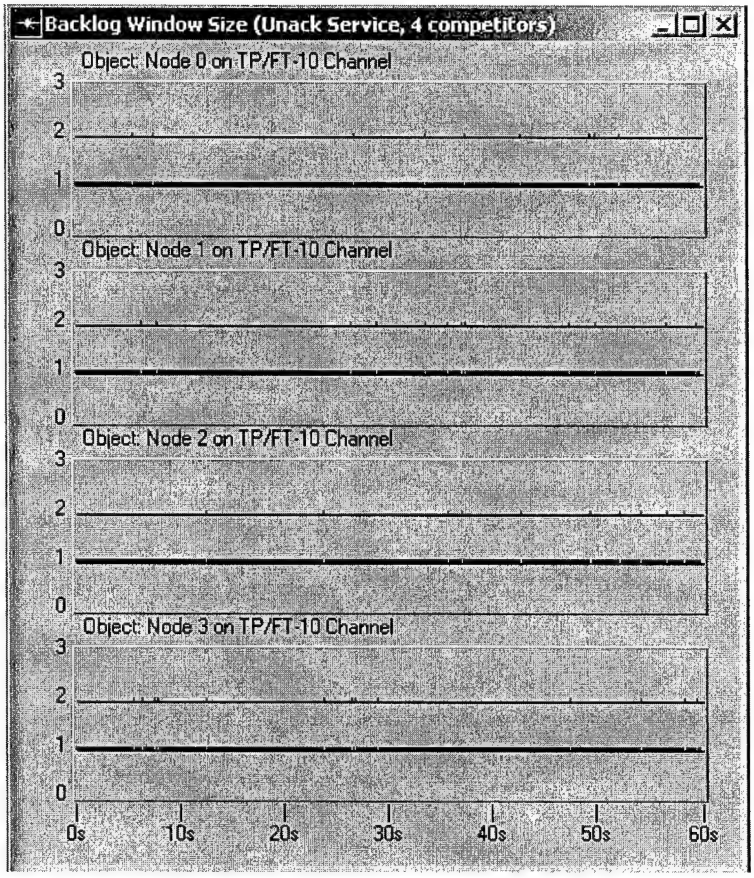


Figure 69 Channel Backlog (UNACK Service)

Figure 69 illustrates the quantity of backlog size changes as time changes for each node. Since we used UNACK service type transmission, the predictive part of the MAC algorithm does not dynamically expand the estimation of backlog size, which causes the backlog size of each node to remain at 1; the node behaves like a simple p-persistent algorithm with $p=0.0625$. We also show the estimation of backlog size for the ACK service in Figure 70. Compared to the figure of UNACK service, we notice that the estimate of backlog size is dynamically adjusted throughout the

simulation. The ACK service activates the predictive algorithm, which results in dynamic adjustment the size of the randomizing window size according to the current estimated backlog size *BL*. Because *BL* ranges from 1 to 63, predictive p-persistent CSMA algorithm operates under $p=0.0625$-persistent under light traffic load conditions and $p=0.0099$-persistent in situations of overload. Between these bounds p varies dynamically according to the current load conditions.

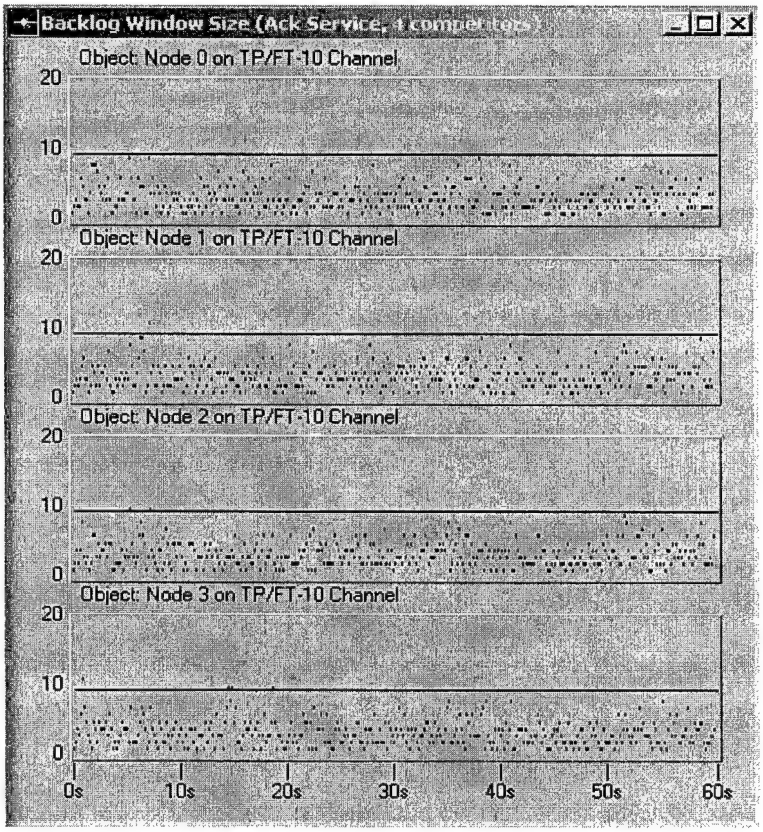


Figure 70 Channel Backlog (ACK Service)

Figure 71 shows the time history of collision rate for every node. We can see that the number of collisions fluctuates between 0 and 10. The analytical prediction (see Figure 63) with four nodes is slightly higher than 10%. The different occurs because the rate of generating packets for each node in our study case (60 messages/second) is lower than the saturation-of-traffic conditions that we used in the analytical model.

Figure 72 shows a simulation results of the end-to-end delays for every node. End-to-end delay is defined as the time between a packet is ready to be transmitted and it is received. Figure 73 shows the number of transmission attempts, Figure 74 shows additional statistics.

7.5 Conclusion

We have developed a communication simulator based on OPNET, which describes LONWORKS network communication behavior. The simulator assumes that the nodes are characterized in terms of basic communication parameters, and does not use specific realization of actuators and

controllers. Comparison with other tools and physical measurements demonstrate satisfactory prediction performance of the simulator.

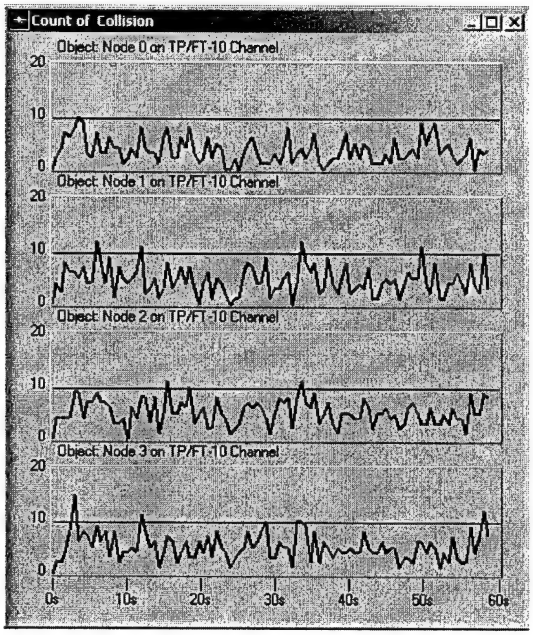


Figure 71 Counting of Collision (UNACK Service)

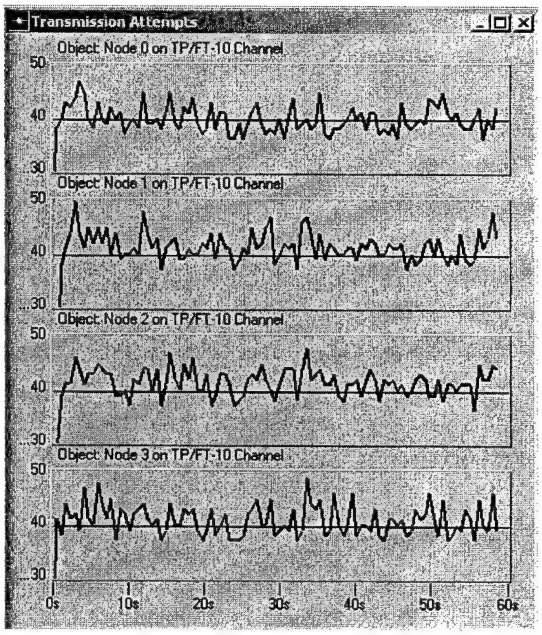


Figure 73 Transmission Attempts (UNACK Service)

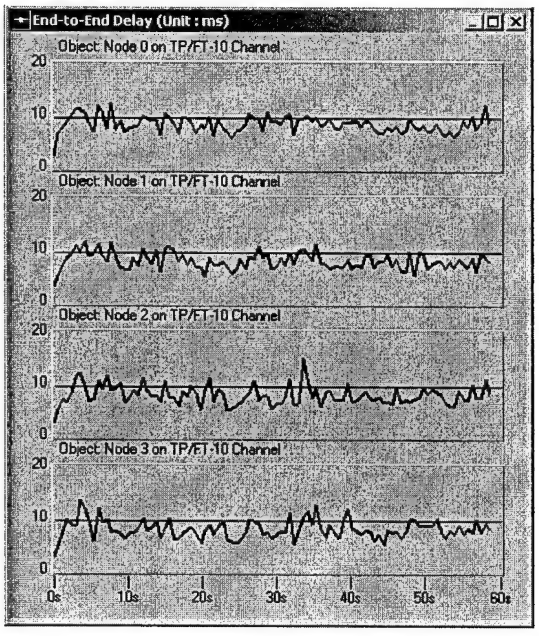


Figure 72 End-to-End Delay (UNACK Service)

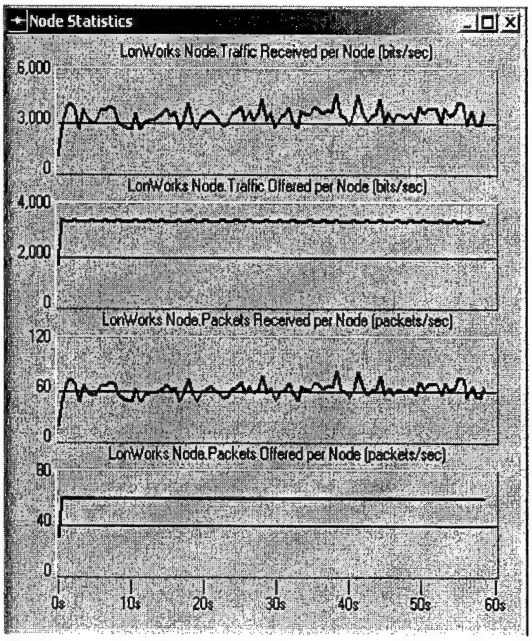


Figure 74 Additional statistics (UNACK Service)

8. CONCLUSION AND OUTLOOK

At the conclusion of Phase II the sponsor possesses a Network Simulator prototype for full-scale simulation of distributed control architectures. The sponsor also possesses a Communication Simulator for rapid comparison of distributed network designs at a high level of abstraction. Both simulators are suitable for networks with few scores of nodes, and were validated on small and medium scale physical systems, ranging from 4 to 108 nodes.

As an immediate short-term extension of the project, we recommend a dissemination and education effort that will demonstrate and test the simulation prototypes with Navy practitioners. This effort would require extended documentation and teaching materials, pilot one-on-one training and demonstration sessions, and several full-scale workshops to Navy engineers.

The primary objective of a third phase of the project is to provide a simulation tool that can handle large-scale distributed control systems involving up to 1000 nodes. We expect this jump to high node counts to challenge the computational efficiency of our software design. It is likely that a single desktop PC will not provide enough computing power to solve the problem in a reasonable amount of time. We may need to attack the problem with several multi-processor machines, and engage in parallel processing.

In Phase III, we also plan to extend user-interface capabilities to process directly Neuron-C code developed for LonWorks nodes. Our simulation environment will accept the same code that a design engineer writes for a hardware target. We also plan to extend the tool's capabilities beyond LonTalk, providing support for additional industrial networking protocols (possibly Profibus, Foundation Fieldbus, or FAIRNET).

In the final phase of the project we will also introduce *automated testing facilities* to perform sensitivity analyses and search for the "weakest link". The tool will allow the design engineer to perform directed searches to find input scenarios that lead to instabilities.

9. REFERENCES

1. Cisco (2002): Ethernet Technologies, on-line: http://www.cisco.com/univercd/cc/td/doc/ cisintwk/ itodoc/ethernet.htm . Accessed April 2004.

2. Echelon (2004): Lonworks development tools, on-line: http://www.echelon.com/products/development . Accessed March 2004.

3. "LonTalk Protocol", *LonWorks Engineering Bulletin,* 1993

4. Echelon (2003): Neuron C Programmer's Guide, Revision 7, Echelon Corporation Document 29300, San Jose, CA, 2003. Available on-line: http://www.echelon.com/support/documentation/Manuals/078-0002-02G.pdf .

5. Fairmount Automation (2002): Development of a Virtual Distributed Control System (VDCS) Test Platform (Phase I), Final Report, Phoenixville, PA 2002.

6. LonMarks (2004): LonMarks home webpage, on-line: http://www.lonmarks.org Accessed March 2004.

7. Microsoft (2002): Microsoft COM technologies, on-line:http://www.microsoft.com/com/tech/ com.asp , last updated March 30, 1999. Accessed March 2004.

8. Motorola LonWorks Technology Device Data Book.

9. ANSI/EIA-709.1-A, *Control Network Protocol Specification*, 1995

10. X. Chen, G.-S. Hong, A simulation study of the predictive p-persistent, CSMA protocol, *Proceedings of 35th Annual Simulation Symposium*, pp. 345-351, 2002

11. OPNET Modeler, OPNET, http://www.opnet.com/products/modeler/home.html

12. G. Bianchi, Performance Analysis of the IEEE 802.11 Distributed Coordination Function. *IEEE Journal on Selected Areas in Communication,* Vol. 18, No. 3, March 2000

APPENDIX I : DEVICE AND PLANT APPLICATION DEFINITION FOR CODE GENERATION AUTOMATION

CONTENT

Application Program Conversion

Scheduler Simulation

Timer Expiration Handling Functions

Network Variable Definition

Network Variable Propagation

Plant Simulation Process

I/O Definition

Network Interface Update Functions

The LonTalk Network Simulation Configuration Tool automates the process of LonWorks network simulation generation. The network device application programs are written in Neuron C. Unfortunately, due to its requirement of specialized APIs and compilers, the Neuron C language cannot be supported directly in the simulation. A translator is employed in this situation to convert the user specified application program into the format that satisfies the requirement of the simulation and a standard C/C++ compiler. In addition to supporting the application program, the translator is also used to process the simulation code for the plant and to embed the simulation code into the simulation tool supported structure. This document defines the format and implementation of the application translation and simulation.

Application Program Conversion

The LonTalk network simulation is developed in C++. The application program simulation code is required to comply with that format and be implemented as C++ classes. The translator will translate the application programs and compose them into a simulation code compatible format. The devices that have the same applications will be defined as objects using the same application program class. The translator converts node application programs in NC format to a unified format called *temporary* file that is supported by the LonTalk network simulation API. The converted code is reconstructed and composed into the target code. The *target* code of the translator contains a header file `appgm.h` and a source file `appgm.cpp` in C++. An include.h file is also constructed to hold the included header files. The user defined network variable types in the NodeBuilder Resource Editor are stored in a user.h file.

The maximum number of application programs in a network simulation is defined as *N*. Each of the application programs is defined in a class. Thus there are *N* application

program classes defined in the `appgm.h`, no matter how many application programs are actually specified by the user. They are identified by the number *n* at the end of the class name (where *n* is a number from 1 to *N)*. An unused class is defined as a class that has no member variables and no member functions. But they have one constructor with the same arguments according to the simulation requirement.

```
class CAppgmn  : public CDevice
{
public:
     CAppgmn(IDeviceAPI * pIAPI,uint8 uid);
}

CAppgmn::CAppgmn(IDeviceAPI * pIAPI,uint8 uid)
     :CDevice(pIAPI,uid)
{}
```

The translator converts the application program and fills the translations into the empty classes. In the header file, where the classes for the simulated application programs are declared, variables and functions are declared as protected members in a class. Every global variable including the defined constant, timer and network variables are defined as the members of the corresponding class. The data type definitions are also included in this class. The following summarizes the conversion of different parts of code in Neuron C.

1. Include Files

   A header file `include.h` is introduced to hold all the included files in the device application NC source code. The `include.h` file is included in the device application simulation header file. The translator creates the `include.h` file. When scanning each NC files, the translator detects the included files and adds them into the `include.h` file.

2. Defined Constant

   In different application programs, defined constants may have different values for the same constant name. The simulation code integrates all the application programs into a single file. Therefore, this format results in a redefinition problem. To avoid potential problems caused by the redefinition, the defined constant is replaced by the numeric constant.

3. Data Type Definition

   A data type definition problem may occur if the same data type name is defined as different data types or structures. To maintain the definition of the data type, the data type is defined in the application program class before the declaration of any variables.

The Neuron Chip is an 8-bit system, while the simulation is run on a 32-bit Wintel system. Their data type definition is different. To fix the difference, we convert the Neuron C data type to the appropriate data type in a Wintel system that has the same type and size. However, enumerated data types are an exception since they are defined as integer types according to ANSI. Unfortunately, an integer has different size in the two systems. It is impossible to change the data type of enumerations in the current development IDE VC++. However, the latest MS .Net C# does provide a mechanism to customize the size of enumeration data types.

The NC data type is converted in the simulation according to the following table. Those data types not listed in the table are not converted.

unsinged	long	int	->	uint16
unsigned	long		->	uint16
unsigned		char	->	uint16
signed	long	int	->	int16
	long	int	->	int16
unsigned	short	int	->	uint8
unsigned	short		->	uint8
unsigned		int	->	uint8
signed		char	->	int8
		char	->	int8
signed	short	int	->	int8
signed		int	->	int8
	short	int	->	int8
		int	->	int8
unsigned	s32_type		->	uint32
	s32_type		->	int32

Table 1. Data Type Conversion Table.

4. Initial Value

Variables and timers usually are assigned an initial value in the application program. The initialization of the data in the simulation is a little different due to the implementation in a class. The variables are defined in the class and initialized in the class constructor. For the variable without an initial value, a value 0 is assigned to it. For the user may assume a default value of 0. Some variables, such as a timer variable, are initialized in the function to reset the node. Some variables may be assigned a value in the application program or by a network management message update. The initialization in these cases is discussed later.

5. Network Variables

A network variable is defined in the translated code in the same way as a common variable. The initial value of network variables is set to 0 unless another value is specified in the application program.

The network variable array is defined as one variable. But they are connected as individual variables and each has its own selector. According to the implementation of the selector assignment, each element of a network variable array is defined separately in the translated code and added to the node using the `AddNV()` function individually. The first element definition is the same as the network variable array, using the array size. However the remaining elements should be defined as common network variables, following the first element.

When a network variable is updated in the application, it will be propagated to the channel during the critical section that immediately follows the variable update. If a network variable is updated more than once in the application it is only propagated once. In the application, a flag is introduced for each network variable. Whenever a network variable is updated, the corresponding flag is set. A function `PropagateNV()` is used in a critical section to propagate the network variables with a set flag .

6. Timer

The timer is defined as `MsTimer` or `sTimer` representing a millisecond timer and a second timer respectively. The timer variables are initialized using the `SetMsTimer()` or `SetTimer()` functions. A timer is usually initialized in a reset function. However a timer may be initialized when declared. The function `TimerEvent()` is introduced to handle timer expiration events and to restart repeating timers. The `TimerEvent()` function detects the expiration of a timer and sets the timer expiration event. For repeating timer, the function also resets the timer to its initial value. The application simulation repeatedly calls this function to update timers. The API function `timer_expires()` will look up the timer expiration events to determine whether a timer has expired. These functions are described later.

7. EEPROM

If any variable is defined as `eeprom` storage class, it is not initialized in the application program but by the network management tool. The values of this kind of variables are stored in the EEPROM of a Neuron chip and preserved even while the node is powered down. Variables defined in `eeprom` storage class usually are used to store a node application configuration. A network variable configuration tool can be used to modify these variables. The network configuration tool can display the value of a network variable and make modification to it. The user can use the network configuration tool to assign values to `eeprom` variables.

8. `when` Statements

The main body of the application program in NC is formed by `when` statements. This part of the NC code is translated and placed in a function called `Run()`. NC `when` statements are translated to `if` statements and switch statements in the C++ version to implement the scheduler.

Example: Neuron C Conversion

The Original Code:

```
#include "fileA.h"
#define Constant_A    20
#define Constant_B    3.75

typedef struct
{
      .
      .
      .
}TypeA;
TypeA var_A;
bool  eeprom var_B;
network output  int        nvo;
network input SNVT_temp_f  nvi;
mtimer repeating sample_timer = Constant_A;

when (Condition_A)
{
      Code_A;
}
when(Conditon_B)
{
      Code_B;
}

void Function_A()
{ Code_C; }
```

The Translated Code:

```
class Appgmn  : public CDevice
{
public:
      CAppgmn(IDeviceAPI * pIAPI,uint8 uid);
      ~CAppgmn();

protected:
      typedef struct
```

```
      {
            .
            .
            .
      }TypeA;

      void Function_A();

      TypeA var_A;
      bool  var_B;
      uint8 nvo;              bool b_nvo;
      float nvi;
      Mstimer sample_timer;

      void Init(NVDefinition *pNvdef);
      void Run();
      void PropagateNV();
      void TimerEvent();
}

CAppgmn::CAppgmn(IDeviceAPI * pIAPI,uint8 uid)
: CDevice(pIAPI,uid)
{}

void CAppgmn::Init(NVDefinition * pNvDef)
{
      CDevice::Init(pNvDef);

      m_nvDef[0].nvLength=sizeof(nvi);
      m_nvdef[0].varAddr= pNVArray[0]=&nvi;
      b_nvo = false;
      …
      memset(&var_A, 0, sizeof(var_A));
      memset(&var_B, 0, sizeof(var_B));
      memset(&nvo, 0, sizeof(nvo));
      memset(&nvi, 0, sizeof(nvi));
      …
      sample_timer = 20; // Constant_A is replaced by its
value.
      …
}

CAppgmn::Run()
{
      WHEN_RESET {}

      START_WHEN(Condition_A)
      {
            Code_A;
      }
      WHEN (Conditon_B,2)
      {
```

```
            Code_B;
      }
      …

      END_WHEN
}

void CAppgmn::PropagateNV()
{
      if (b_nvo) {
            Propagate(&nvo);
            b_nvo = false;
      }
      …
}

void CAppgmn::TimerEvent()
{
      if(MsTimerExpired(&sample_timer))
      {
            expired_timer[0]=& sample_timer;
            SetMsTimer(&sample_timer,
sample_timer.setTimerValue);
      }
      …
}

CAppgmn::Function_A()
{
      Code_C;
}
```

Scheduler Simulation

There are two kinds of when clauses the scheduler manages. The first one is priority when, the other is the common when clauses. The translation of the when clauses from Neuron C is specified in the previous section. Refer to the examples for translation details.

For priority when clauses, the clause that appears first in the code always has a higher priority than those that appear below it. (See Neuron C Reference Guide for more information.) The scheduler for a priority when statement and a reset event statement is implemented using `if...then` statements as follows:

```
if (reset) {
      code 0;
}
else if (condition1)
      code 1;
}
```

```
.
.
.
```

For common when clauses, the scheduler works in a round-robin style and each when clauses competes each other. The scheduler checks the conditions of all the when clauses in the application program. Each time only one of the when clauses whose condition is satisfied can be executed. Thus the scheduler implementation uses switch statement.

```
else {
      switch (count) {
      case 0:
            count++;
            if (condition2) {
                  code 2;
                  break;
            }
      case 1:
            count++;
            if (condition3) {
                  code 3;
                  break;
            }
            .
            .
            .
      case n:
            count++;
            if (condition4) {
                  code 4;
            }
            count = 0;
      }
}
```

The above code can be made more readable and made to look more like the original Neuron C code by using macros. These macros are defined below.

```
#define WHEN_RESET                if (reset)
#define PRIORITY_WHEN(condition)  else if (condition)
#define START_WHEN(condition)     else { switch(count) { case 0: count++; if (condition) {
#define WHEN(condition, n)        break;} case n: count++; if (condition)
{
#define END_WHEN                  } count = 0; }}
```

Using the macros to replace the long and tedious code in the previous example produces a more succinct and much more readable format for the programmer. The converted code is listed below.

```
WHEN_RESET {
```

```
        code 0;
}
PRIORITY_WHEN(condition1) {
        code 1;
}
        .
        .
        .
START_WHEN(condition2) {
        code 2;
}
WHEN(condition3, 1) {
                code 3;
}
        .
        .
        .
WHEN(condition4, n-1) {
        code 4;
}
END_WHEN
```

Timer and Expiration Handling Functions

Timers are used in the device application to keep track of time. The timer implementation belonging to a given thread, for example an application program or the stacks, is updated based on the global simulation time rather than the local clock. The global simulation time evolves according to all the updates of the local simulation clocks of the threads. Different threads are executed at different time points in terms of simulation time. Thus the evolution of the global simulation time is generally finer than any given thread making smaller time increments with each step. Since the simulation can take care of a timer expiration event once at a clock update event, and a timer may expire at any time during the interval of a simulation clock updates, clearly setting or checking the expiration of a timer according to a timer with finer granularity results in a high accuracy simulation. The simulation guarantees that every time the simulation time gets updated the timers are checked.

The simulation provides functions such as `SetMsTimer(MsTimer *timerOut, uint32 valueIn)` and `MsTimerExpired(MsTimer *timer)` to set a timer or check the timer state. The `void TimereEvent()` function uses these functions to update a timer and to check its expiration event in the device application simulation. If a repeating timer expires, this function starts the timer again.

The Neuron C code has an event called `timer_expires[(timer-name)]` to check a timer expiration event. To support this feature, the simulation tool provides an array `MsTimer *expired_timer[80]` for each device to store the address of each expired timer. The upper bound number of timers in a device is 80, which is the size of the array.

We may use dynamic memory allocation to optimize the size of this array later. The translator will read the application program to determine how many timers the application uses and generate a function called `void TimerEvent()`. The format of this function is

```
void CAppgmn::TimerEvent()
{
      if (MsTimerExpired(&timer1)) {
            expired_timer[0] = &timer1;
            // timer1 is a repeating timer.
            SetMsTimer(&timer1, timer1.setTimerValue);
      }
      if (MsTimerExpired(&timer2)) {
            expired_timer[1] = &timer1;
      }
      .
      .
      .
      if (MsTimerExpired(&timerk)) {
            expired_timer[k-1] = &timerk;
            // timerk is a repeating timer.
            SetMsTimer(&timerk, timerk.setTimerValue);
      }
}
```

`timer1`, `timer2`, ... and `timerk` are the different timers; `setTimerValue` is the assigned value of a timer. If a timer is a repeating timer, the timer needs to be re-started again whenever it expires. If the timer is not a repeating timer, the line `SetMsTimer(&timerk, timerk.setTimerValue)` is not needed.

The simulation API provides two functions to handle the timer expiration event. They are

```
Boolean timer_expired() {}
Boolean timer_expired(MsTimer &timer);
```

The first function corresponds to the unqualified `timer_expires` event. The function will search through the `expired_timer[]` array to see if there is an entry with a valid address having an expired timer. If so, the function returns `true`. Otherwise, the function returns `false`.

The second function is to replace the qualified `timer_expires` event. The function checks the expiration of a given timer. The function will look for the entry of `expired_timer[]` with the same address as the given timer. If an entry is found, the function clears that entry and returns `true`, otherwise it returns `false`.

Network Variable Definition

The network variables are defined in the application program. The automation tool needs to read the definitions in the network design and makes it available to the simulation tool.

Thus it is necessary for the automation tool to search the network variable definitions and import them into the network definitions. The automation tool searches the network variable defined in the application program and all its attributes in the LNS database, and then stores the definition in the file `NV.FMT`. The format of the file is

```
//Subnet id
//Device id
priority,direction,selector_hi,selector_lo,turnaround,service
,authen,0,0,0,0x80,0x30,0,0,0,0,"Out","OutSD",NULL,addr_index
,
… …
//Device id
… …
```

The NV definition starts with the subnet id and device id. These ids define the device the following NV definition belongs to. A NV definition includes priority, direction, selector number, turnaround flag , service type, authentication flag, and address table index. The other values in the middle of the NV definition are not used. The flags, such as priority, turnaround, ad authentication, use 1 for true and 0 for false. The direction flag uses 1 to represent outgoing NV and 0 for incoming NV. Refer to the `nv_struct` definition in LonWorks Technology Device Data book and `NVDefinition` defined in GadgetStack API Functions Reference for more details.

Network Variable Propagation

When a network variable is updated within a when clause in the application code, the propagation of this NV is scheduled after the present critical section. In other words, the NV is not sent out immediately, but waits until the Neuron Chip finishes execution of the user application program. In simulation, the way a NV is propagated is imitated by introducing a flag for each NV, indicating if propagation is required or not.

When translating the application source code. A flag is inserted in the code.

```
int  NVO;  bool b_NVO;
```

The flag is set to true at every place an update of this NV occurs.

```
NVO = value;     b_NOV = true;
```

A function is introduced for a device to propagate all the NV after the critical section. The function `void PropagateNV()` is defined as

```
void PropagateNV()
{
      if (bNVO) {
            Propagate(&NVO);
            BNVO = false;
      }
      .
```

```
        .
        .
}
```

This function is defined as the member function of a device application class.

Plant Simulation Process

Similar to the application program, the LonTalk simulation supports more than one plant simulation. In addition, we assume that each plant runs by itself but it is possible that different plants share the same simulation code to simulate identical physical plants. In the simulation we limit the total plant simulation to a constant *M*. But the total number of plant simulations does not have an upper bound. Each of the plants is defined in a class. Therefore, no matter how many plants are actually specified by the user, there are *M* plant classes defined in the `application.h` header file. They are identified by the number *m* at the end of the class name (*m* is a number from 1 to *M*). An unused class is defined as a class that has no member variables and no member functions. But they have one constructor with the same argument according to the simulation requirement.

```
class CPtpgmm: public CPlant
{
public:
      CPtpgmm(IPlantAPI * pIAPI,uint8 uid);
};

CPtpgmm:: CPtpgmm(IPlantAPI * pIAPI,uint8 uid)
 : CPlant(pIAPI,uid)
{}
```

If the user defines a plant and provides the simulation code, the complete plant simulation class looks like the following:

```
class CPtpgmm : public CPlant
{
public:
      CPtpgmm(IPlantAPI * pIAPI,uint8 uid);
//    ~ CPtpgmm();

protected:
      bool Init();
      bool UnInit();
      bool Run();

      //define variables here.
};

CPtpgmm:: CPtpgmm (IPlantAPI * pIAPI,uint8 uid)
      : CPlant(pIAPI,uid)
{}
```

```
bool CPtpgmm::Init()
{
      fDeltaT = (double)…;//the simulation update period.
      // initialize other variables here.

      Return TRUE;
}

bool CPtpgmm::UnInit()
{
      // clear data here.

      return TRUE;
}

bool CPtpgmm::Run()
{
      // the simulation code.
}
```

The format of the plant simulation source file should be defined with the following format.

```
// include header files
#include "includedFile.h"

// define constant
#define const_1;

// define plant simulation variables
double     state_1;
double     state_2;

.
.
.

// some other initialization code.

// define plant simulation
bool main()
{
      // simulation code.

      return TRUE;
}

// define un-initialization function.
bool UnInit()
{
      // clear memory allocation.
```

```
        Return TRUE;
}
```

I/O Definition

1. Device I/O object and Access Functions

   I/O object is defined for up to 12 I/Os: `IO_0`, ..., `IO_11`. These I/O objects are associated with each of the individual I/O values of the device simulation. I/O functions are defined to access these I/O objects:

   ```
   float io_in(IO);
   void io_out(IO);
   ```

   The parameter `IO` in the functions are any of the I/O objects. A float value is returned when calling the `io_in(IO)` function. If the I/O is defined as a digital I/O, the returned value is either `1.000`, or `0.000`.

2. Plant I/O and Access Functions

   The plant I/O values are defined in the network configuration tool. A variable is assigned to each I/O and the input/output type is also specified for the I/O. Two I/O access functions are defined:

   ```
   float GetInput(const  uint8 &idx), and
   void SetOutput(const float &data, const uint8 &idx).
   ```

   They are used for input and output respectively. For input function, the returned value updates the related variable. For output functions, the `data` parameter refers to the related variable. The parameter idx in both functions are the index of the buffer entry in the network interface, where the I/O data is buffered when sent between a device and a plant. The indexes are defined in a configuration file `io.xls`.

3. SmartControl I/O Definition

   The selection of the network device is limited to Smart Control devices. The IO implementation is developed as a generic model, so that the implementation of different devices is very similar. There are a total of 11 I/Os available for Smart Control devices. The definitions of those I/Os are:

Index	Name	Type	Corresponding Function	Function Parameter
0	IO_1	Relay	io_out(relay1, RELAY_ON)	relay1
1	IO_2	Relay	io_out(relay2, RELAY_ON)	relay2
2	IO_3	Relay	io_out(relay3, RELAY_OFF)	relay3
3	IO_4	Digital I/O	N/A	N/A
4	IO_5	Digital I/O	N/A	N/A
5	IO_6	Analog Output	adr11x_da(0,data)	0
6	IO_7	Analog Output	adr11x_da(1,data)	1
7	IO_8	Analog Input	adr11x_ad(0,data)	0
8	IO_9	Analog Input	adr11x_ad(1,data)	1
9	IO_10	Universal Input	adr11x_ad(2,data)	2
10	IO_11	Universal Input	adr11x_ad(3,data)	3

Table 1. Smart Control I/O Definition.

IO_1 to IO_11 are I/O names. Refer to the *User's Manual for Smart I/O™ ADR110 and ADR112 Programmable Multifunction I/O Modules* for the I/O definition. Each I/O is associated with an index value from 0 to 10. In the simulation the indices are used as the identification of an I/O. The translator will interpret the I/Os that the I/O functions interact with in terms of these indices. The next section will describe in detail how the interpretation is handled.

Network Interface Update Functions

When the control network simulation includes a plant simulation, the interaction between the plant and the devices use the network interface. The interface has a float type array. Both the output of plant simulation and device simulation are fed into the entries of the array. The inputs of the plant simulation and device simulation also read values in these entries. In the simulation code the network interface can be accessed in an application program as:

```
m_pNetwork->m_fData[]
```

For the device simulation, 12 variables are defined corresponding the 12 I/Os in a device. The defined variables are `float fVar[0-11]`. The examples of update I/O are listed here:

1) Update `fVar[1]` with the value of the 6th entry in `m_fData`

```
m_pNetwork->m_fData[5] = fVar[1];
```

2) Update input `fVar[5]` from 7th entry in `m_fData`

```
fVar[5] = m_pNetwork->m_fData[6];
```

A synchronization algorithm that runs at the highest level of the network simulation governs the simulation program's execution of plants and devices. In both a plant simulation and a device simulation program the I/O is updated based on the synchronization algorithm. The I/O updates described above are defined in the following two functions:

```
void UpdateInput()
void UpdateOutput()
```

The two functions are defined as a member function of both the `CPlant` and `CNodeApp` in the simulation code. The first function updates the inputs of the device and the second function updates the output of the device. The translator will write the related I/O update code into these two functions. The location of the `UpdateInput()` and `UpdatOutput()` are depend on the synchronization algorithm. When the synchronization algorithm decides to update the I/O, these two functions are called respectively.

The configuration tool gathers the interface configuration and generates a file `IO.xls`. The `IO.xls` file specifies the I/Os of every device and plant, and designates the entry of the buffer array for each I/O to read/write. The I/O update functions use this file to determine the indexes of connected ports and update the I/Os in the way as shown in the examples above.

APPENDIX II : DEVICE APPLICATION SIMULATION API

back

The LonTalk Network Simulation supports network device applications. The device application simulation is designed in such a way that the user can debug their device application(s) during a simulation. However the simulator only supports C++ style code. In the LonTalk Network Simulation Configuration Tool, an NC-to-C Translator is used to translate a NC application program to the C++ style code. To make the translated device application code readable, the LonTalk Network Simulation provides an Application Programmer Interface (API), which supports most important NC program features. Therefore, the translated device application may still keep the NC format, which is familiar to the user.

1. Implementation Structure

The simulation code can simulate multiple network devices in one processor. The application programs of all the simulated devices are implemented in a thread. A simulation controller synchronizes the threads to execute their simulation tasks according to the simulation time evolution of each thread.

The device application simulation implements a set of functions for a device application. These functions are listed below:

```
void Init(NVDefinition *pNvdef);
void Run();
void PropagateNV();
void TimerEvent();
```

The function `void Init(NVDefinition *pNvdef)` is called during the device simulation initialization. In this function, the network variable definitions are loaded and variables and timers are initialized according to the device application code.

The function `void Run()` executes the device application. The thread calls this function every time the simulation controller activates it. As the result of the function call, the simulation time of the simulated device will increase based on the time cost for the simulated event.

The function `void PropagateNV()` propagates network variables that are updated in the device application. In a LonWorks device, when a network variable is updated or assigned a new value, it will be propagated to the channel during a critical section immediately following the variable update. When a network variable is updated, it is not propagated immediately, but waits until the current iteration of the device application finished. The function `void PropagateNV()` is included in the critical section that follows the `void Run()` function to simulate the network variable propagation.

The function `void TimerEvent()` maintains the application timers. In this function, the timer is updated according to the current simulation clock, and timer expiration is handled. If a timer is defined as a repeating timer, the timer will reset to its initial value. This timer function is called more frequently than the others in a thread.

2. Data Type

Data types are supported in the configuration tool in a way similar to NC. Data types related to the API are defined based on the network variable type definition in the *Neuron C Programmer's Guide Revision 7*. Standard network variable types are also supported, as well as user defined data types. The user defined network variable type or data type may be defined in the same file where the device application is defined. If a type is developed using the LonBuilder Resource Editor, a manual input is necessary to define the type again in a file and include this file as a header file in the device application file. During a translation process, the data type in Neuron C is converted to the appropriate data type in C++.

3. Scheduler

The scheduler executes the application program in response to the condition specified in the `when` clauses. The simulated scheduler assumes the default scheduling algorithm is applied. There are three types of when clauses: the special when clauses, the priority when clauses, and the non-priority when clauses.

The special when clauses have the highest priority and are taken care of immediately. These events include reset, offline, wink, etc. When clauses with these events are checked first. If any event is true, the first one in the list will be executed. After executing the task, the scheduler goes back to the beginning.

The priority when clauses are checked after the special event when clauses are checked. The scheduler handles the priority when clauses the same way as the special event when clauses in that the first event that evaluates to true is executed first. Therefore, the implementation of the special event when clauses and priority when clauses is coded with `if...else if...else` constructs. For example, the following Neuron C code can be implemented into the code that follows it.

```
when (reset)
{
      task1;
}
priority when (condition1)
{
      task2;
}
priority when (condition2)
```

```
{
      task3;
}
```

The scheduler implementation for this code is:

```
if (reset)
{
      task1;
}
else if (condition1)
{
      task2;
}
else if (condition2)
{
      task3;
}
```

For non-priority when clauses, the scheduler works in a round-robin style and each when clauses competes with the others. The scheduler checks the conditions of the when clauses in the application program. Each time only one of the when clauses whose condition is satisfied can be executed. Thus the scheduler implementation uses a switch statement. For example, a code with a non-priority when clauses is defined as following:

```
when(condition2)
{
      code 2;
}
when(condition3)
{
      code 3;
}
.
.
.
when(condition4)
{
      code 4;
}
```

The simulation code for the scheduler to implement the source code above is:

```
else {
      switch (count) {
      case 0:
            count++;
            if (condition2) {
```

```
                code 2;
                break;
            }
        case 1:
            count++;
            if (condition3) {
                code 3;
                break;
            }
            .
            .
            .
        case n:
            count++;
            if (condition4) {
                code 4;
            }
            count = 0;
        }
}
```

Further, the awkward code can be beatified with macros to make it more similar to the original Neuron C code. The macros are defined below.

```
#define WHEN_RESET                if (reset)
#define PRIORITY_WHEN(condition)  else if (condition)
#define START_WHEN(condition)     else { switch(count) { case 0: count++; if (condition) {
#define WHEN(condition, n)              break;} case n: count++; if
(condition) {
#define END_WHEN                        } count = 0; }}
```

Using the macros to replace the long and tedious code in the previous code examples produced a succinct and much more readable format for the programmer. The converted code is listed below.

```
WHEN_RESET {
    code 0;
}
PRIORITY_WHEN(condition1) {
    code 1;
}
    .
    .
    .
START_WHEN(condition2) {
    code 2;
}
```

```
WHEN(condition3,1) {
            code 3;
}
      .
      .
      .
WHEN(condition4,n) {
      code 4;
}
END_WHEN
```

4. Events

Most conditions are based on certain events. All the events fall into five classes. They are system wide events, input/output events, timer events, message and network variable events and user specified events.

In the application program interface, only the timer events and message and network variable events are supported. (Some system-wide events may be supported in the future; I/O events may also be developed in the future.) The user-specified events are supported as long as the Boolean expression is available and valid.

4.1 Timer Event

The functions `bool timer_expires()`, `bool timer_expires(mtime timer)` and `bool timer_expires(stimer timer)` are defined to check timer expired events.

The function `bool timer_expires()` returns true if any of the defined timers expire. It is equivalent to the NC event `timer_expires`. The functions `bool timer_expires(mtimer timer)` and `bool timer_expires(stimer timer)` check the specified timer and return true when the timer expired. Once an expired timer is checked, the expiration flag of that timer is reset.

The function `MsTimerExpired(MsTimer *timer)` is used to check the expiration of a timer. `SetMsTimer(MsTimer *timerOut, uint32 valueIn)` can start a timer by setting the timer value. This function can be used to initialize a timer in an initialization function. It is used in the function `OnTimerEvent()` to reset a repeating timer when the timer expires.

4.2 Message and Network Variable Events

The supported message and network variable events are:

```
nv_update_occurs
nv_update_fails
```

```
nv_update_succeeds
nv_update_completes
msg_arrives
msg_complete
msg_succeeds
msg_fails
resp_arrives
```

All these events are updated automatically in the simulator. So of these events are available for the application program directly. However the event functions with the same name as the events above are not supported, for example, `msg_arrives(uint8 code)` and `msg_succeeds(MsgTag tag)` are not available.

5. Functions

The following functions are supported in the API to process explicit messages.

```
msg_alloc()
msg_alloc_priority()
msg_free()
msg_receive()
msg_send()
```

`msg_cancel()` is declared but not implemented.

The response/request functions listed next are also supported in the API.

```
resp_alloc()
resp_cancel()
resp_free()
resp_receive()
resp_send()
```

For the explicit message and request/response messages, the following variables are declared.

```
msg_in;
msg_out;
resp_in;
resp_out;
```

For network message pooling function `poll` is supported as

```
poll()
poll(int16 nvIndexIn)
```

The `poll()` is equivalent to the Neuron C function `poll()` without an argument. The `poll(int16 nvIndexIn)` is same as the `poll(network-var)` in Neuron C.

For timer handling, the API provides three functions:

```
SetMsTimer(MsTimer *timerOut, uint32 valueIn),
UpdateMsTimer(MsTimer *timerInOut), and
MsTimerExpired(MsTimer *timer).
```

`SetMsTimer(MsTimer *timerOut, uint32 valueIn)` initializes the timer `timerOut` to `valueIn`. `UpdateMsTimer(MsTimer *timerInOut)` updates timer `timerInOut` based on the present time. `MsTimerExpired(MsTimer *timer)` checks the current value of the timer and determines whether it is expired. If the timer is expired, the function return value is true, otherwise it returns false.

6. I/O functions

The I/O object is defined for up to 12 I/Os: `IO_0`, ..., `IO_11`. These I/O objects are associated with each of the individual I/Os of the device simulation. I/O functions are defined to access these I/O objects:

```
float io_in(IO);
void io_out(IO);
```

The parameter `IO` in the functions are any of the I/O objects. A float value is returned when calling the `io_in(IO)` function. If the I/O is defined as a digital I/O, the returned value is either `1.000`, or `0.000`.

When using I/O objects and functions, the plant simulation is assumed to define in the network simulation; and the used I/O objects are connected to the Plant I/Os. If these device I/O objects are not connected to a plant I/O, the reading from an I/O is the initial value of the I/O, which is 0 by default. Other I/O related functions and events are not implemented in the API.

7. Application Response Speed

The device application on a network device runs at a variable speed. The speed depends on various aspects, such as the application computation load, number of when statements, number of timers, and number of network variables. The timer event, network variable update events, etc. are handled during the critical section between two iterations of the device application. The application response time determines the response frequency that the events are checked, which is critical for network performance.

The dynamics of the device application is difficult to simulate in a generic manner. Nevertheless, the response time may be adjusted for each individual application based on testing or estimation. The function `void SetDeltaT(double t)` is introduced to change the response time. This function may be put into the device application code wherever there is a need to change the default response time. Neuron C does not support this function. Remember to remove this function before compiling the application.

The response time change effected with this function is only valid for the current simulation step. The response time will be reset to the default value after the current simulation step completes. To gain high accuracy, this function may be inserted in every `when` clause. An effective alternative way is to use repeating timers to control application activities and set both the timer and the response time, so that the timer value is longer than the response time.

back

LonWorks® Network Analysis Toolkit User Guide

Version 1.0.2

Revision 1

Fairmount Automation Inc.

April 2004

Table of Contents

1. Introduction

2. Getting Started

System Requirement

Hardware Requirement
Supporting Software

Installation and Initialization

Installing the Toolkit
Initialization

3. Designing a LonWorks® Network

Design Overview

Creating a Network

Creating a Device
Creating a Channel
Creating a Router

Creating Network Applications

Creating a Device Application
Special Requirement for Neuron C Code
Format File

4. Simulating a Network

Building a New Network Simulation

Managing Network Simulations

Open a Simulation
Close a Simulation
Save a Simulation
Delete a Simulation

Operating a Network Simulation

Starting a Simulation
Pausing a Simulation
Stopping a Simulation

Configuring Network Variables

5. Using the Analysis Tools

Network Traffic Monitor

Network Traffic Analyzer

Channel Analyzer

Overall Statistics
Device Bandwidth Statistics
Device Message Count
Message Classification Statistics

Message Log Window

6. Debugging Device Applications

Device Application Simulation

Debugging Device Applications

7. Advanced Topics

Application Response Time

More about IP Channels

Analysis Tool Dependency

Appendix A Neuron C Device Application Template

Appendix B Software Architecture and Dependency

1. Introduction

The LonSim LonWork® network analysis toolkit is a design tool for network simulation, network traffic visualization, and analysis. It helps the network designer to evaluate network message schemes based on a network design using LonMaker 3.0 or more advanced versions. The software is designed to be accurate, intuitive, easy to use, and user friendly.

The software accesses the LNS database to retrieve the network design and build the network simulation automatically. The software provides network traffic analysis tools for network performance evaluation and diagnosis. These

capabilities are provided at the network level, as well as at the channel and device levels.

These analysis tools enable the user to monitor the throughput and bandwidth utilization of a channel, to identify the design bottlenecks, to identify vulnerable devices or routers, to diagnose channel performance, and to trouble-shoot messaging schemes and device applications.

The features of the software are:

1. Builds simulation scenarios automatically from a network design created with standard design tools (e.g., LonMaker).
2. Supports any network topology and size of network compliant with the EIA709.1 standard.
3. Provides simple interface to modify network designs and start network simulations.
4. Provides a simulation progress clock and mechanism to run, pause, and stop simulations.
5. Stores results of simulation runs for future review.
6. Provides comprehensive network traffic analysis tool to monitor network traffic.
7. Supports multiple network analysis tools to monitor different channels at the same time.
8. Provides a Network Traffic Monitor to compare channel traffic.
9. Provides a Network Analyzer to analyze network traffic details within a channel and between channels.
10. Provides a Channel Analyzer to evaluate performance of a channel and its devices.
11. Provides a Message Log Window for packet monitoring and recording.

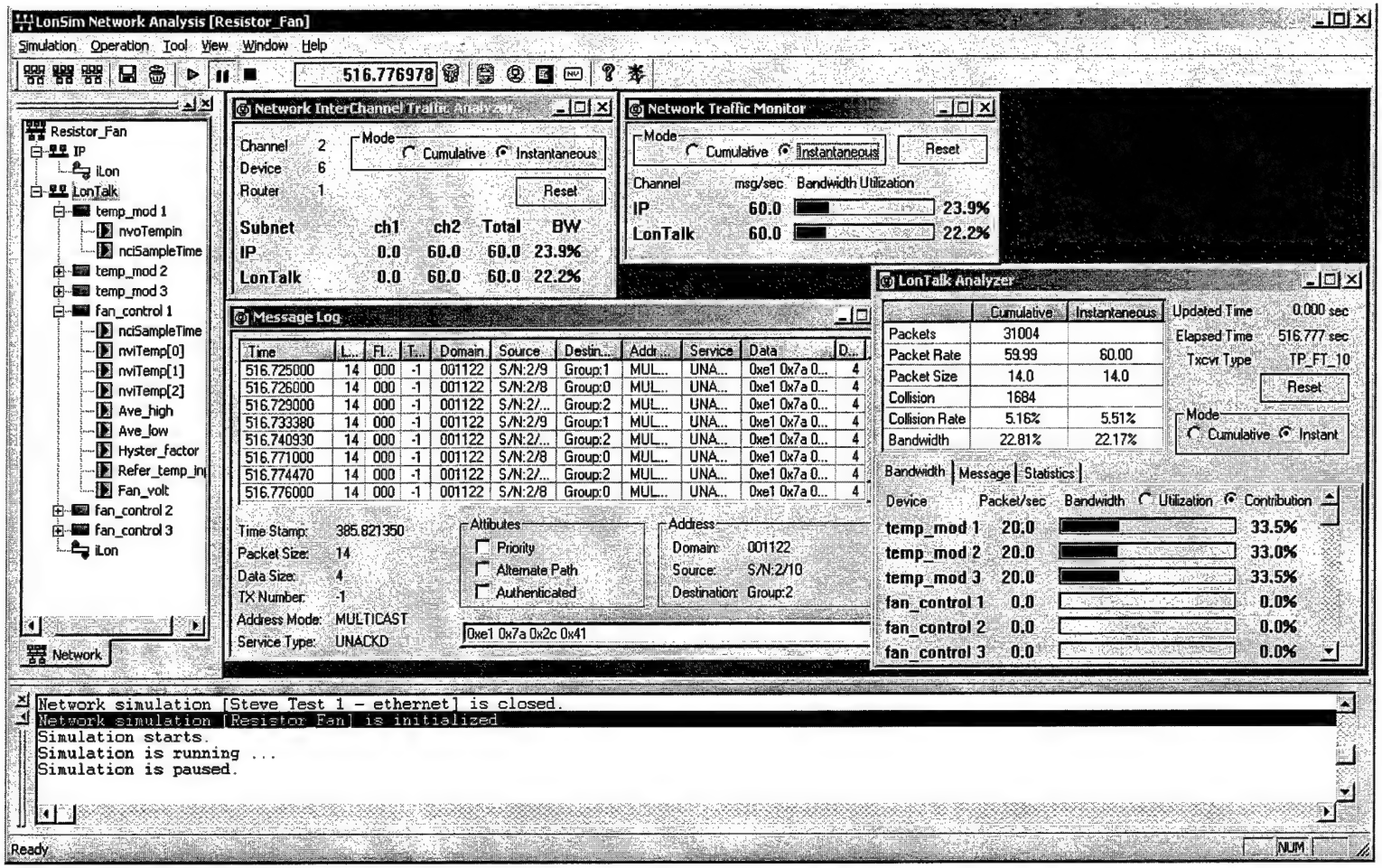


Figure 1.1. LonSim LonWorks Network Analysis Toolkit.

2. Getting Started

System Requirement

Hardware Requirement

- Windows 2000 is recommended.
- Minimum 800 MHz CPU. 1 GHz AMD processor is recommended.
- Minimum 256 MB RAM. 512MB is recommended.
- Minimum of 50 MB free memory.

Supporting Software

The following third party software is required

- LonMaker 3.0 or a more advanced version.
- NodeBuilder 3.0 or a more advanced version.
- Microsoft Visual Studio 6.0 or a more advanced version.

Refer to Appendix C for the Software Architecture and Dependency.

Initialization

For Windows 2000/NT:

1) Click **Start** from the taskbar.
2) Select **Settings** in the Start menu.
3) Click **Control Panel** from the submenu.
4) Open the **Display** in the Control Panel.
5) The **Display Properties** dialog is shown. Select the **Effects** tab.
6) In **Visual effects** check the **Show windows while dragging**.
7) Click the **Apply** button and then click the **OK** button.

3. Designing a LonWorks® Network

Design Overview

Generally, the LonWorks network design procedure should follow the LonMaker manual and the Neuron C Programmer Guide.

Creating a Network

The general requirement of a network design is to comply with the LonMaker user guide. The user designs the network topology, specifies the network application, and binds the network variables. Since no physical network is attached, it is not necessary to commission any devices.

The network design must meet the following requirements:

Only network variable messages are supported in the simulation. Connection services are supported in a limited manner. This means that the user needs to set the transaction timer, receiving timer, repeat timer, retry counter, and repeat counter, for advanced connection services such as repeating unacknowledged messaging, acknowledged messaging, or group messaging. Do not set any timer or counter to automatic.

Creating a Device

All network devices are required to be programmable devices with user defined application code written in Neuron C. Refer to next section to creating Network Applications, for the requirements for device applications. When creating devices, do not commission the device. (This network design tool will not consume any LonWorks credits.)

Pre-programmed devices, such as LonPoint devices are not supported. In the event that simulation of a LonPoint device is desired, use a programmable device and provide a device application program in Neuron C that mimics the functionality of the LonPoint device.

Creating a Channel

The Network Analysis Toolkit supports TP/FT-10, TP/XF-1250, IP-10L and IP-10W channels.

There should be no device directly connected to an Ethernet channel. To simulate an Ethernet channel with devices, add a LonWorks channel that connects to the Ethernet channel. Add the Ethernet devices to the LonWorks channel. To avoid burdening the channel in a heavy traffic situation, add one LonWorks channel for each Ethernet device.

Creating a Router

All routers are assumed to be configured routers.

Creating Network Applications

Creating a Device Application

The device application must be created in Neuron C using NodeBuilder. The toolkit assumes the source code is located in the default directory specified by NodeBuilder, namely where the project files are stored. NodeBuilder compiles the device application and generates an .XIF file. The .XIF file is used to define the device application in LonMaker. The toolkit uses the path of the .XIF file to locate the NC source file by default. If the default source file is not found, VDCS will query the user to specify the source file. The toolkit uses the source code directly to simulate a device, instead of the image files.

Special Requirements for Neuron C Code

The toolkit only supports limited features of Neuron C. Thus the device application code should be programmed according to the Neuron C Application Code Template. The template describes all the features the simulation supports, and should be strictly followed in developing device applications. The template is described in Appendix A.

For the current version, only network variables are permitted for communication. Explicit messaging is not supported. Network variable types defined in the NodeBuilder Resource Editor should be defined again in the user.h file in the DeviceApp folder. The user code of the device application needs to be in one file.

Format File

The toolkit supports network variable configuration to display and overwrite network variable values. Please refer to Network Variable Configuration in Chapter 4. The NV configuration requires a format file to correctly represent the data of a network variable.

The format for standard network variable types and common data types are already defined in VDCS_SNVT.fmt. The user needs to provide the format file for user-defined types. The format file specifies the name of a network variable type and its display format. A network variable type that is already defined in the VDCS_SNVT.fmt file should not be defined in the user-defined format file. For example, if there is a network variable type UNVT_TYPE defined as:

```
typedef
```

```
{
    int a;
    int b;
} UNVT_TYPE;
```

The entry in the format file for UNVT_TYPE will be:

```
UNVT_TYPE:        text("%d %d");
```

Please also refer to the NB_SNVT.fmt file for the format of a format file. This file is located in \LonWorks\types\ directory.

The default format file name is <network name>.fmt. The default location of the format file is in the root of the network directory. For example, there is a network named MyNet, and it uses the NodeBuilder project MyNetApp. There is a folder MyNet in the directory: \Lm\Source\MyNet. The application is in the folder \Lm\Source\MyNet\MyNetApp\. The format file MyNet.fmt should be in the folder \Lm\Source\MyNet\. If the format file is not there, the toolkit will show a warning when building the network simulation. In this case, the network variable configuration for the network variables with the user-defined type will not work correctly.

4. Simulating a Network

Building a New Network Simulation

The user may begin building network simulations with the toolkit once the network and application designs are complete. To begin the process, run the LonSim.exe application. The toolkit will open as shown in Figure 1.1 but without any analysis tools. The only available command options for the user are "New Simulation", "Open Simulation", and "Delete Simulation".

The user can push the "New Simulation" button in the tool bar or select the menu item "Simulation > New..." to begin a new simulation. The New Network Simulation dialog shown in Figure 4.1 will be displayed for the user to choose the network for simulation.

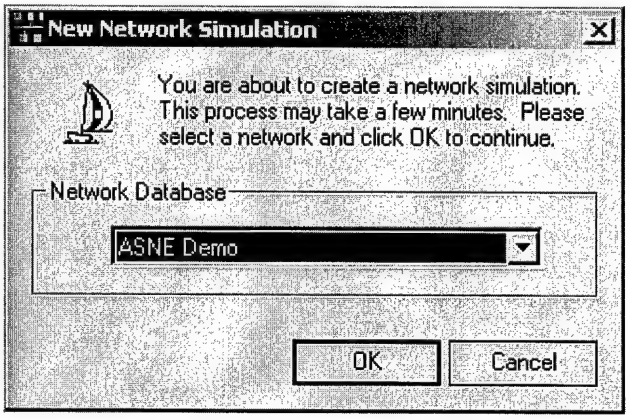


Figure 4.1. New Network Simulation Dialog.

As shown in Figure 4.1, all the LonWorks networks in the LNS database are listed in the Network Database combo box. Select the network to be simulated and press the "OK" button. The network simulation process will start. A progress window will appear that states "preparing the network simulation ... " and displays progress information.

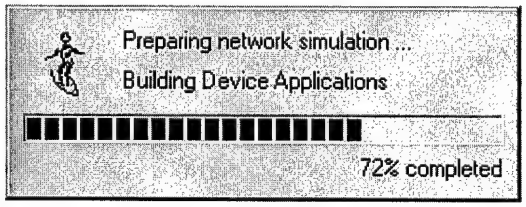


Figure 4.2. Progress Dialog.

During the simulation building process, the software opens the LNS database of the network specified in the New Network Simulation Dialog, and extracts network definition information including network device templates, network topology, network connections, etc. If the software cannot locate the application NC source files, it will prompt the user to locate them. All the information is saved into a single data structure. Then the network configuration files are created based on this information. At the same time, the NC2C translator scans the network application source files to create network application simulation files and related application configuration files.

In the final stage, the software launches the MSDEV application in MS Visual Studio to compile the network application simulation files using a predefined project file. The compilation information is shown in the message pane. The toolkit checks the progress of the build process and the status of the compilation. Once the compilation finishes, the new simulation is opened automatically.

Managing Network Simulations

The user can build a new simulation, open an existing simulation, save the active simulation or deleted a saved simulation.

Opening a Simulation

To open a saved network simulation, use the "Open Simulation" button on the toolbar or the "Simulation > Open..." menu item. The Open Network Simulation dialog shown in Figure 4.3 will be displayed. Click the network to be opened, and the network name will appear in the Network Name box. Then press the OK button to open the selected network simulation.

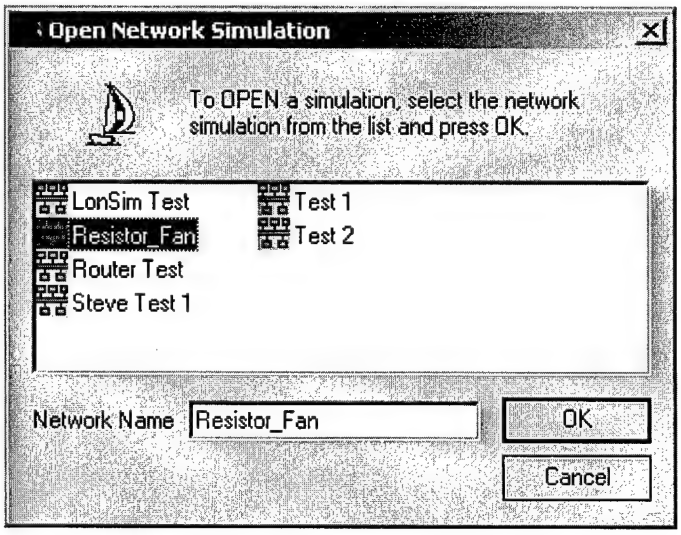


Figure 4.3. Open Network Simulation Dialog.

Closing a Simulation

Use the "Close Simulation" button on the toolbar or select the menu item "Simulation > Close..." to close the network simulation. This will release the current loaded simulation. It is not necessary to close the simulation before building a new simulation or closing the program, as the simulation will be closed automatically in those cases.

Saving a Simulation

When saving a simulation, click the "Save Simulation" button on the toolbar or use the menu item "Simulation > Save..." The Save Network Simulation dialog shown in Figure 4.4 will be displayed. The style is the same as the Open Network Simulation dialog. This function saves the current active network simulation and is only available when a network simulation is active or opened.

If you try to save a network with the same name as a previously saved network simulation, you will be prompted with an option to overwrite the existing file or change the simulation name.

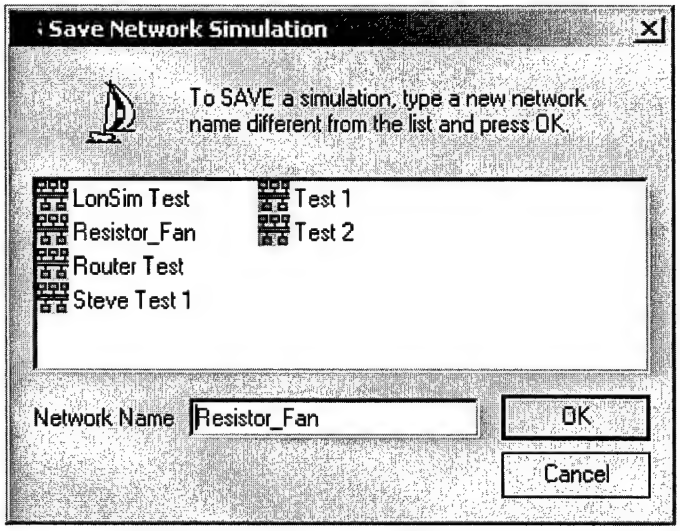


Figure 4.4. Save Network Simulation Dialog.

Deleting a Simulation

When deleting a network simulation, use the "Delete Simulation" button on the toolbar or select the "Simulation>Delete..." menu item. The Delete Network Simulation dialog shown in Figure 4.5 will be displayed. Click the network to be deleted and push the OK button.

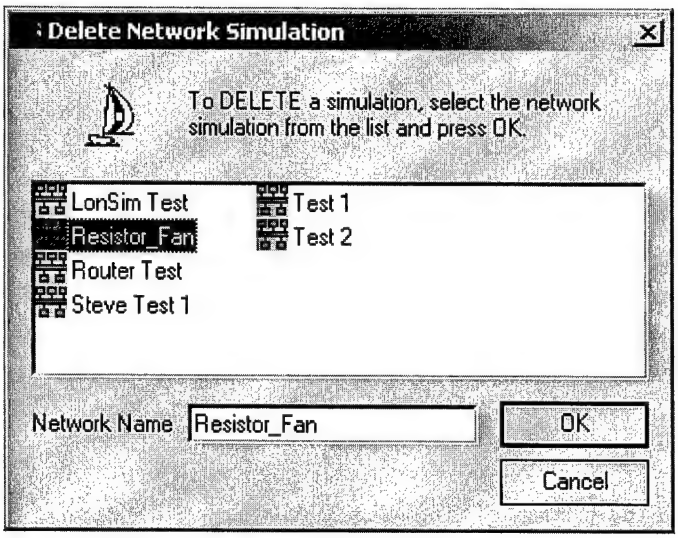


Figure 4.5. Delete Network Simulation Dialog.

Operating a Network Simulation

Once a simulation is open, the user can start the simulation, stop a simulation, or pause a simulation. These commands are available in the toolbar and also in the Operation menu. They are enabled after opening a network simulation.

Starting a Simulation

The network simulation is initialized when the user presses the "Start" button on the toolbar. The simulation starts immediately. The simulation progress clock is displayed in the toolbar.

Pausing a Simulation

The "Pause" button pauses a simulation. When a simulation is paused, the simulation stops progressing and the CPU computation load drops to zero. This makes it easy for the user to run another heavy-computation process during a simulation. Pausing a simulation will not affect the simulation result. The simulation will resume when the user presses the "Start" button.

Stopping a Simulation

The "Stop" button stops a network simulation. After a simulation is stopped it remains loaded in memory and can be re-executed. However, the simulation is completely re-initialized. The stop function allows the user to start a simulation over again.

Configuring Network Variables

The toolkit provides a network variable configuration tool to display and modify network variable values. A format file is necessary to display network variables in a recognizable format and to write new values to it correctly. For standard network variable format, refer to the SNVT.fmt or NB_SNVT.fmt files provided by LonWorks in the folder: \LonWorks\Types\. For network variables of user-defined types, a user defined format file must exist in the folder of the designed network. Refer to the Format Files section in Chapter 3 for format file requirements.

Use the "NV Configure" button on the toolbar or select the "Tool > NV Configuration..." menu item to open the NV configuration dialog shown in Figure 4.6. To display a network variable, select the channel, device and network variable. To configure a network variable, the new value should keep exactly the original format. Then press Configure button to update the value.

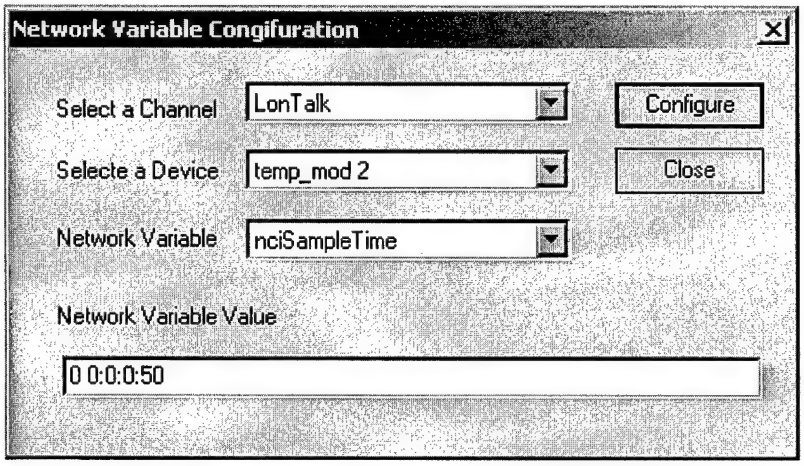


Figure 4.6. Network Variable Configuration Dialog.

The network variable configuration tool emulates the LonMaker Browser used to modify a network variable. The LonMaker Browser sends a network management message to a device to update a network variable. Similarly, the network configuration tool puts a message event into the device event queue when a network variable is updated. Thus, to avoid overflowing the queue, only press the Configure button once.

5. Using the Analysis Tools

Four tools are available for network traffic analysis at different levels. The Network Traffic Monitor displays general traffic loads throughout a network. The Network Traffic Analyzer checks the traffic flow between channels as well as the traffic load of each channel. The Channel Analyzer monitors a channel's general traffic statistics and the activities of each device. The Message Log records all the messages in every channel.

The simulation does not provide statistics for an Ethernet channel. Although Ethernet channels are shown in the workspace and analysis tools, the statistical data displayed is not valid.

Network Traffic Monitor

The Network Traffic Monitor provides the most general traffic information. It helps the user to check the overall network traffic. It is the only analysis tool automatically opened when a simulation is opened.

As shown in Figure 5.1, it displays the packet throughput and bandwidth utilization of each channel. Two modes are selectable: cumulative and instantaneous. The cumulative mode shows the statistics starting from the beginning of the simulation. The instantaneous mode shows current statistics compiled over a 2-second period of time. The Reset button allows the user to restart the statistics, which clear all the data history.

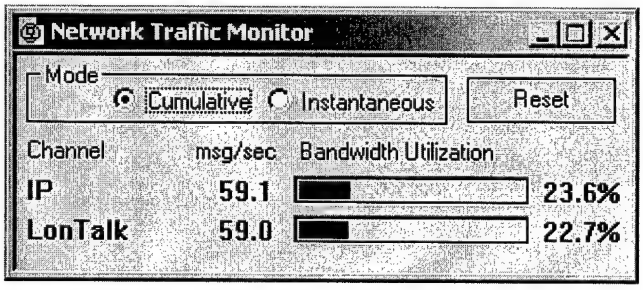


Figure 5.1. Network Traffic Monitor.

Only one Network Traffic Monitor window is allowed to be open at one time. If the Network Traffic Monitor window is closed, you can open it using the "Tool > Traffic Monitor" menu item.

Network Traffic Analyzer

The Network Traffic Analyzer is used to analyze the traffic flow between channels in order to identify over-loaded routers and to observe the direction of traffic flow. It displays each and every traffic source of a channel and the corresponding message volume. Use the Network Traffic Analyzer button

in the toolbar or use the "Tool > Network Analyzer" menu item to display the dialog box shown in figure 5.2. Only one Network Traffic Analyzer is allowed to open at one time.

The Network Traffic Analyzer displays the number of channels, number of devices, and the number of routers in a network. Similar to the Network Traffic Monitor, the Network Traffic Analyzer has two selectable modes and a Reset button. The selection between the two modes determines the statistic data displayed in the window. The Reset button clears the statistic history. It will reset all the statistics displayed in any open Channel Analyzer windows as well.

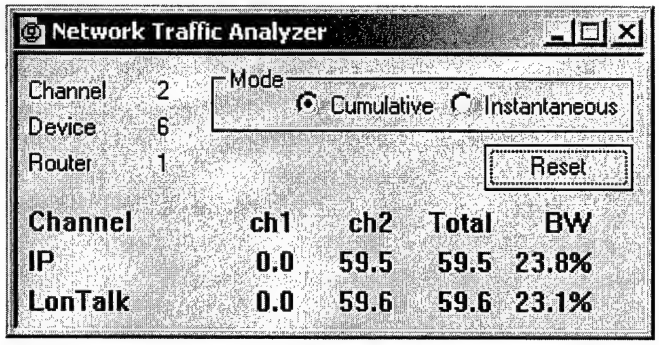


Figure 5.2. Network Traffic Analyzer.

The lower part of the Network Traffic Analyzer shows the traffic information. Each row displays the traffic flows into a channel, and other general information. The first column from the left is the channel name. The first column from the right is the bandwidth utilization of a channel. The second column from the right is the total packet throughput. In the middle there are packet throughput data from neighboring channels, as well as the total throughput generated by the devices in a given channel. The channel names are not listed for each column in order to provide a compact format. Instead the labels ch1, ch2, ... are used. From left to right, they each correspond to the names in the left first column from top to bottom.

For example, as shown in Figure 5.2, the value 0.0 in the first row and first column is defined by IP and ch1. Since ch1 means channel IP, the value 0.0 is the throughput of devices in channel IP. The value in the second column of the same row is defined by IP and ch2. The ch2 represents the channel named LonTalk. The 59.5 value in that cell represents the throughput of the router connected to channel LonTalk. The sum of these two values is the total throughput of the channel IP.

Channel Analyzer

The Channel Analyzer provides a more detailed view of channel traffic activities. A Channel Analyzer has three parts as shown in Figure 5.3. To the

left is the channel general traffic statistics. To the right are the simulation time, transceiver type, reset button and mode controls. At the bottom of the Channel Analyzer are the advanced analysis tools.

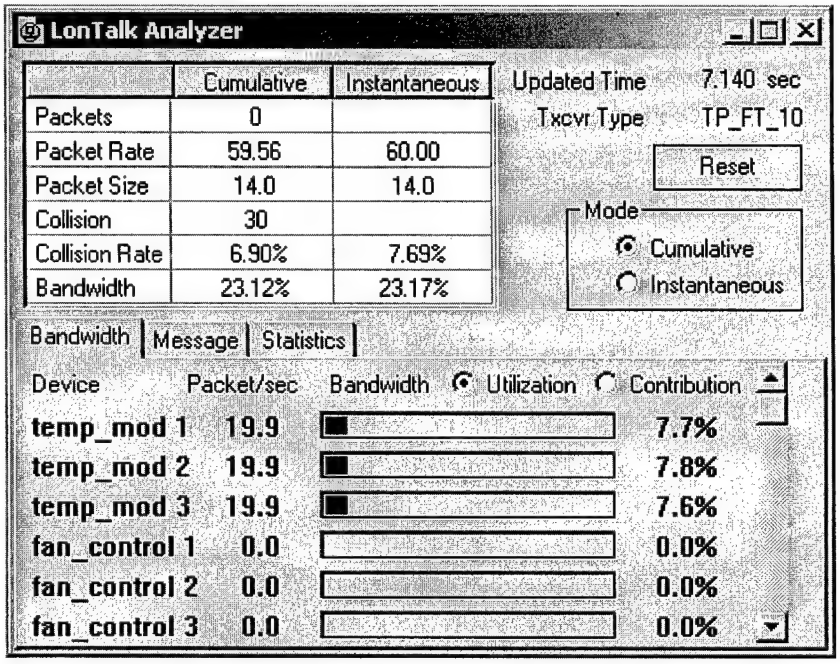


Figure 5.3. Channel Analyzer.

There are three different ways to open a Channel Analyzer. One method is to use the menu item "Tool > Channel Analyzer". Push the Channel Analyzer button in the toolbar is another way. When using these two methods a channel selection dialog is open. Select the channel from the combo box and press OK.

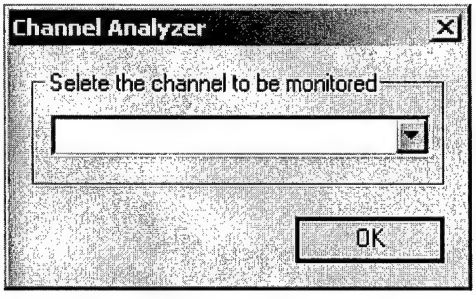


Figure 5.4. Channel Selection Dialog.

A fast way to open a Channel Analyzer is to use the floating menu in workspace. Move the mouse over a channel or select a channel by left clicking the mouse, and then click the right mouse button. A floating menu appears. Then click Channel Analyzer. The Channel Analyzer of the selected channel will be opened.

Overall Statistics

The channel general statistics include cumulative data and instantaneous data. The cumulative data shows the transmitted packet count, the packet throughput, the average packet size, the collision count, the collision rate and the bandwidth utilization. The instantaneous data shows the current packet throughput, the current average packet size, the collision rate, and the bandwidth utilization.

The Update Time is the time when the statistic starts. The txcvr type shows the transceiver or channel media type. It can be TP_FT_10, TP_XF_1250, IP_10L or IP_10W. If the channel type is not supported, Unsupported is displayed. In this case, the simulation is not accurate or correct.

The Channel Analyzer also has two modes and a Reset button. The mode selection works only for the Device Bandwidth Statistics. The Reset button resets all the statistics for the channel. Notice, if a Network Traffic Analyzer is open and the Reset button of a Channel Analyzer is pushed, the related channel statistics in the Network Traffic Analyzer is reset as well. Try to avoid this situation which may make the analysis tricky.

Device Bandwidth Statistics

The Device Bandwidth Statistics tab is one of the advanced analysis tools provided as a part of the Channel Analyzer. It lists packet throughput and bandwidth utilization of all the devices and routers in a channel. A progress bar shows the bandwidth information graphically with a numerical value followed at the end of the same row.

When selecting the Utilization option, the Device Bandwidth Statistics displays the device utilization of the channel bandwidth.

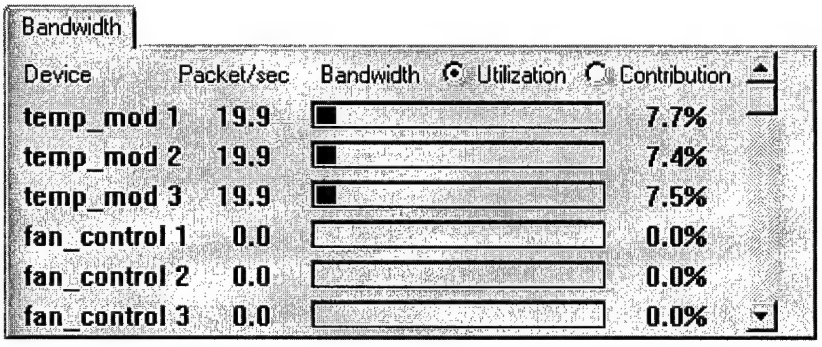


Figure 5.5. Device Bandwidth Utilization.

When selecting the Contribution option, the device contribution to the total channel bandwidth utilization is displayed as a percentage of the overall utilization.

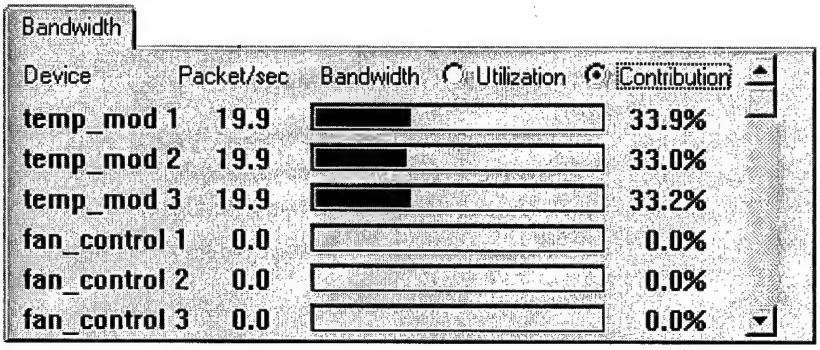


Figure 5.6. Device Bandwidth Contributions.

Device Message Count

The Device Message Count is another advanced analysis tool. It displays the count of scheduled messages in the device stack and the transmitted messages from the transceiver. The lost message count and the lost message rate are also calculated. Since there is a delay between the time when a message is scheduled and the time when it is transmitted, there may be a small number of lost messages shown in the tool, which do not correspond to actual lost messages (they are still working their way thru internal devices queues). This tool helps the user to evaluate the efficiency of the device applications and the channel traffic load.

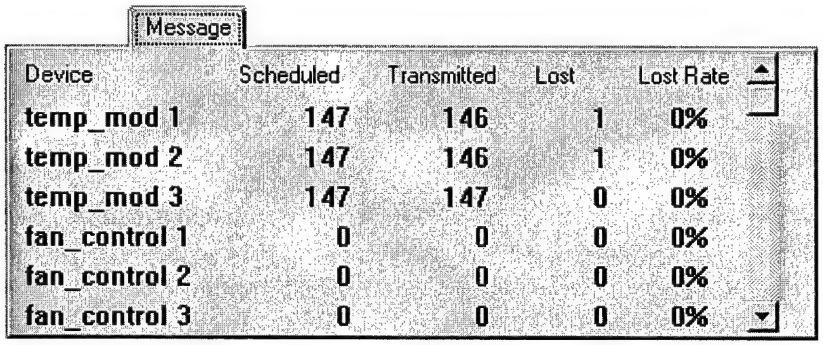


Figure 5.7. Device Message Count.

Message Classification Statistics

The Message Classification Statistics groups the channel messages according to their service type and counts the messages for each group. The messages are classified into 10 groups. The sum of all the counts of each group is the total number of messages transmitted.

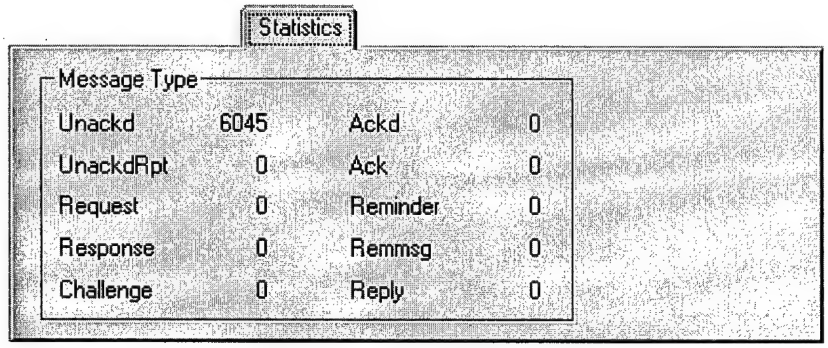


Figure 5.8. Message Classification Statistics.

Message Log Window

The Message Log is an analysis tool which functions as a protocol analyzer logging all network messages. Typically, a protocol analyzer is attached to a single network channel. Only the messages transmitted over that channel are observed. The Message Log Window works differently. It displays the messages from all channels of a simulated network. The messages are displayed in the order of their timestamps (the time when a message appears on a channel). As an exemption, the messages that go through an Ethernet channel are not recorded in the Message Log Window.

Since the Message Log Window records every message throughout a network, it is possible to trace a message as it makes its way thru routers from one channel to another. Propagation delays between channels can be computed by comparing the timestamps of each copy of the message in the Message Log.

To open the Message Log, use the button in the tool bar or select the "Tool > Message Log" menu item. Figure 5.9 shows the Message Log Window.

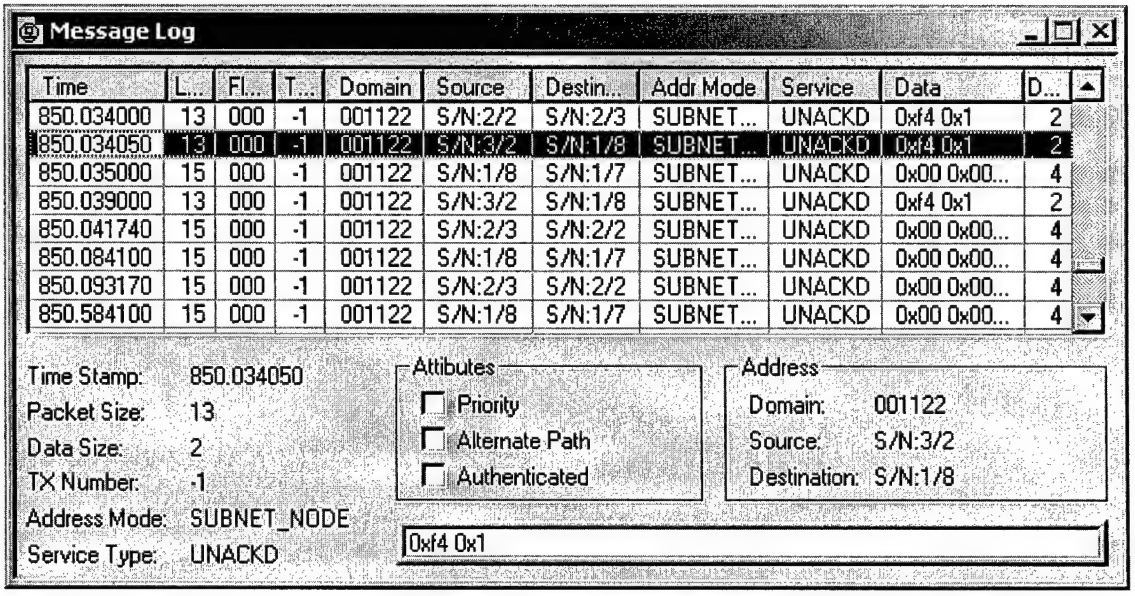


Figure 5.9. Message Log Window

The Message Log Window has two parts. The first part is a list of message record that includes all transmitted messages in the order of their timestamp. There are eleven (11) columns in the list. They are (1) message timestamp, (2) message length, (3) the attribute flags, (4) transaction number, (5) domain ID, (6) source address, (7) destination address, (8) message addressing mode, (9) service type, (10) data, and (11) data size.

The attribute flag has three digits. The first digit is the priority flag; the second one is the alternate path flag; and the third one is the authentication flag. For each digit, a "1" means that the corresponding attribute is true, and a "0" means that it is not. The transaction number is used for messages with acknowledged service. The value is "-1" if there is no valid transaction number in use. The source address is the network address of the source device, in the format of Subnet/Node (S/N). Depending on the address mode, the format of the destination address can be any of the four: (1) Subnet/Node, (2) Group, (3) Subnet, (4) Node ID. The message data shows the hexadecimal value in a small endian order: from left to right with the lowest byte going first.

The lower part of the Message Log Window shows details of a selected message. Select one message by clicking one row in the message log list. The message information will be displayed in the lower part of the Message Log Window.

The message log is recorded as an .xls file. The file name uses the prefix of NV_Log_ and appends the simulation start time to it. For example, if a simulation started at 16:22 on Nov. 25th, 2003, the file name is NV_Log_11_25_2003_16_22.xls. When closing the network simulation, the

following dialog will open providing a choice for the user to save the file or delete it.

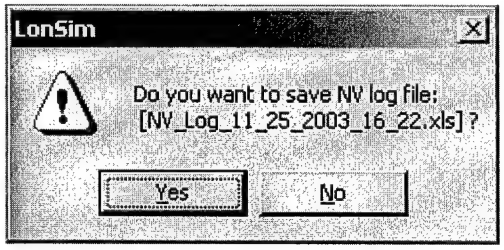


Figure 5.10. Message Log Window Close Dialog.

6. Debugging Device Applications

Device Application Simulation

When LonSim builds the network simulation, the device application simulation is encapsulated in a dynamic link library (DLL) called DeviceApp.dll. In the process of building a device application simulation, the device application code that is programmed in Neuron C is translated into a C++ file before building DeviceApp.dll. The format of the Neuron C file is kept with a minor modifications to allow the user to recognize their code in the C++ device application file. The source code of the DeviceApp.dll is provided for debugging purposes.

In the DeviceApp.dll source code, each device template is implemented in a class. During a simulation, when a device template is used by multiple devices, DeviceApp.dll will implement an object of the class for each device. The simulation will simulate these applications in threads, one thread for a device. Each device uses the unique node id to distinguish each object.

During a simulation, LonSim will drive DeviceApp.dll to simulate the device applications on each network device. DeviceApp.dll works as a server that performs the device application simulation and provides simulation data.

Debugging Device Applications

To debug a network application simulation, the following procedure should be followed:

1. Build a new network simulation with LonSim in debugging mode.
2. Save the network simulation.
3. Close LonSim. (Always close LonSim after a debug simulation is built/opened.)
4. Open the DeviceApp.dsp/dsw project in Visual C++.
5. Start a debug session (F5) to launch LonSim.
6. Open the saved network simulation in LonSim.
7. Open the Device ID Table in the Tool menu to look up the unique node ids.
8. In VC++, set desired breakpoints in the DeviceAppgm.cpp file.
9. Select Edit and open Breakpoints.
10. To debug application of a particular device, set the breakpoint condition to "UNodeID==id"(id is the device unique node id).
11. Run the simulation and debug the application.

Unlike the runtime version, the debug version of a network simulation has to be built every time to debug a network application simulation. Although a debug version of a network simulation is saved, it is only valid for debugging when there is no other network simulation built or opened. Therefore it is a good practice to re-build the debug version simulation every time before debugging a network application simulation, even if the same simulation is saved and there are no changes made to the network.

7. Advanced Topics

Application Response Time

The application response time is the time that an application in a device spends to respond to an event and perform control actions. Any event with a frequency higher than the pre-set response time will not be processed on time. Many application-dependent factors influence the application response time. Therefore it is impossible to precisely predict the application response time of a device.

To simulate a simple device application accurately, it is recommended to use a timer to control output messages. In the simulation, each device has a fixed response time. The pre-set application response time is 10ms. To maintain a high-accuracy simulation, a timer should be set at least to 20ms, which is twice as long as the response time. Any event handling without a timer will be processed at a 10ms interval. Make sure that at any time the device application's response time won't be faster than 10ms or slower than the timer used.

For more complicated applications, an API function is provided to adjust the application response time dynamically. The function is

```
SetDeltaT(const double &time);
```

Add this function at any exit point in the application source code, and provide the time cost for the control action and the scheduler overhead as the parameter. The response time change using this function is effective for the current critical section. The response time will be reset to its default value for the next critical section unless the function is called again. During one critical section, only the last call of this function will actually affect the response time. This function is not supported in NodeBuilder. So before inserting this function into your application code, compile the application first. This function provides the ability to dynamically change the application response time. The user may estimate the response time for each control action or test the response time with the network device.

More about IP Channels

The Ethernet simulation model is based on two assumptions:

1) The time delay for the transmission of a package is considerably shorter than the LonWork router's response time and can be ignored. This remains valid when there is only a small number of routers attached to the Ethernet channel and the traffic on the Ethernet channel is low.

2) The Ethernet channel is used as a high-speed backbone and there are no application devices directly attached to the Ethernet channel.

The simulation does not provide valid statistics for the Ethernet channel. Although Ethernet channels are shown in the workspace and analysis tools, all data are invalid except for the scheduled message count.

However, the scheduled message count for an Ethernet channel in the Channel Analyzer is the total number of messages passed through the Ethernet channel, instead of the count for each router. The transmitted message count is always the same as the scheduled message count. But this does not necessarily mean that all the messages are passed to the LonWorks channel from the Ethernet channel.

Analysis Tool Dependency

Each analysis tool displays statistical data from a particular part of the simulation module. The relationship between the analyzers and the simulation module is shown in Figure 7.1. Some analysis tools refer to the same simulation module, so that there is a dependency between certain analyzers. For example, the channel module in the simulation also contains buffer module for each device attached to it. The channel module and the buffer module all hold a copy of the channel traffic information. The statistical data are generated in these two kinds of modules. When a reset operation occurs, all the data in a module are reset to their initial value. Therefore, if two analysis tools refer to the same module, the reset action from one tool will affect the other.

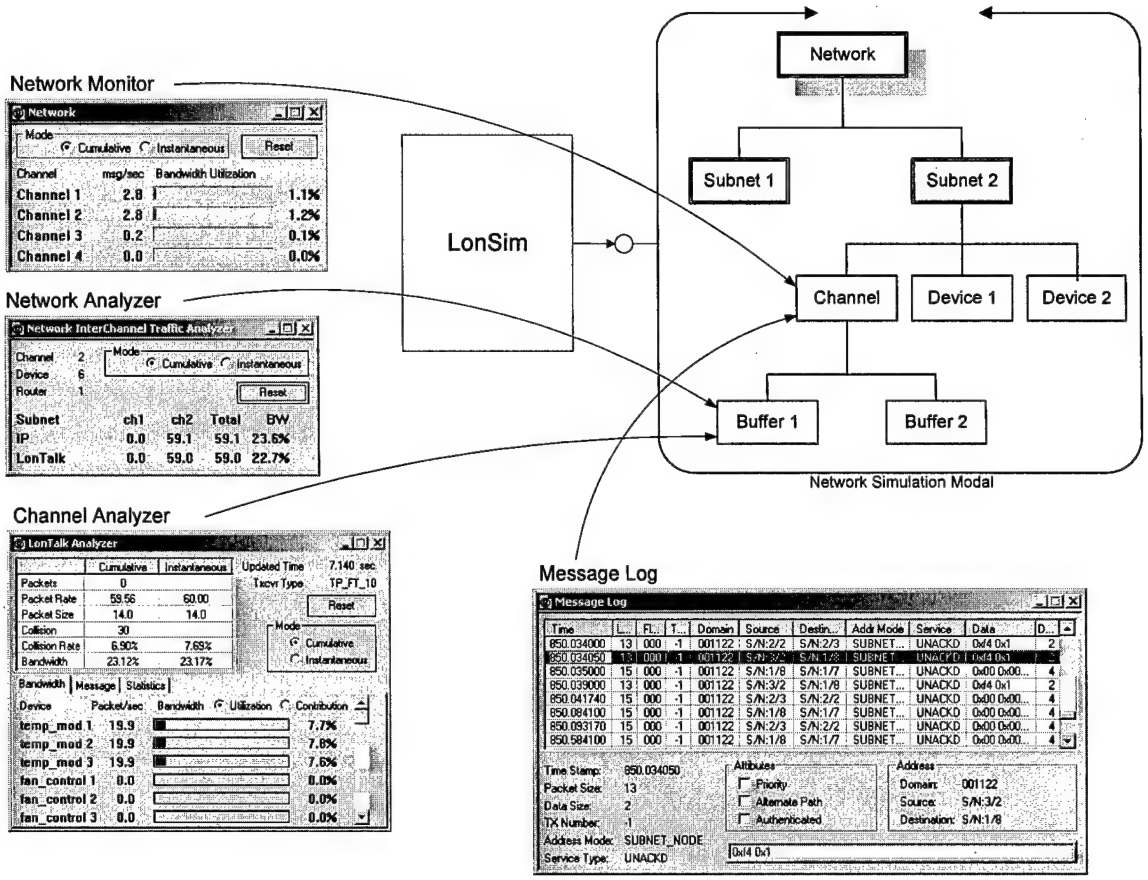


Figure 7.1. Relationship of Analysis Tools and Simulation Objects.

The Network Traffic Monitor and the Message Log Window refer to the data in the channel module. The Network Traffic Analyzer and the Channel Analyzer refer to the buffer module. Since each module has two analyzers referring to it, the reset function of one analyzer will affect the data shown in another analyzer that refers to the same module. This becomes an issue when using the Network Traffic Analyzer and the Channel Analyzer. Once the Network Traffic Analyzer is reset, all the Channel Analyzers are reset too. This may not be a problem. However, when a Channel Analyzer is reset, only the corresponding data in the Network Traffic Analyzer is reset. Therefore, the statistics of different channels in the Network Traffic Analyzer are no longer comparable. We do not suggest resetting a Channel Analyzer when a Network Traffic Analyzer is in use.

Appendix A
Neuron C Device Application Template

```
// vdcs simulation application Neuron C source template.
//
```

```
// Version 1.0.0
// 09. 7. 2003
//


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// Comments
//

// this type of comment is supported.

/*
   this type of comment is supported.
*/


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// constant
//

#define CONSTANT 100


// global variables
//
// any NC data type is supported.
//

int value;        // comment may go here.


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// structure and union
//

typedef struct
{
    unsigned long  day;
    unsigned short hour;
    unsigned short minute;
} time;


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
```

```
//
// network variable definition.
//

network input int nvi;                          //  comment  may
go here.
network output int nvo;
network output int nvo = CONSTANT;
network input int nvi_array[CONSTANT];

// Array network variable is not supported. Even the array
// network variable is defined, the array elements should
// be treated respectively.  Refer to when template 5.

// Explicit message is supported, however address must be //
specified.


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// timer definition
//
// the timer should be set larger than or equal to 10ms.
//

mtimer timer;
stimer timer;

mtimer repeating timer = value;
stimer repeating timer = value;


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// when clauses
//
// Predefined events are not supported except the complex
// event.
// io events are not supported.
//

// when statement template 1.
//

when (reset)
```

```
// when statement template 2.
//

when (expression)


// when statement template 3.
//

when (timer_expires(timer))


// when statement template 4.
//

when (nv_update_occurs(nvi))


// when statement template 5.
//

when (nv_update_occurs(nvi_array[0])||
      nv_update_occurs(nvi_array[1]))


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// function definition
//

void function(void)
{
}


// + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
//
// non-global code template.
//
// ANSI C expressions are all supported for non-global
// expressions. Any one or any combination of the following
// templates can be used in a when statement.
//

// template 1: NV assignment
//
```

```
   nvo = value;


// template 2: timer assignment
//

   timer = value;


// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Illegal Expressions
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// Illegal Expression 1.
//
// Any global expression in multiple lines is illegal.
// Global expression are the code not within any { }.
//

value =
             CONSTANT;


// Illegal Expression 2.
//
// only one declaration is allowed in a line.
//

int value1, value2;


// Illegal Expression 3.
//
// Only one expression is allowed in a line.
//

value1 = CONSTANT; value2 = CONSTANT;


// Illegal Expression 4.
//
// nv_update_occurs can't not check an nv array. refer to
// when statement template 3.
//

when(nv_update_occurs(nvi_array))
```

```
// Illegal Expression 5.
//
// no static variable should be used.
//

static int value;


// Illegal Expression 6.
//
// Predefined events are not supported.
//

when(offline)


// Illegal Expression 7.
//
// I/O operation is not supported.
//

io_update_occurs(io_1);


// Illegal Expression 8.
//
// un-qualified event is not supported
//

when(timer_expires)                    // or
when(nv_update_occurs)


// Illegal Expression 9.
//
// function definition should be in the same line.
//

void
func(void)

// Expression 10.
//
// { } should not be omitted.
//
```

```
while(i < 0)
   i++;
```

Appendix B
Software Architecture and Dependency

The Figure C.1 shows the software architecture and the relationship between the toolkit and the supporting components and software. The GUI of LonSim launches the Simulation Core together with the Device Application to perform the simulation. Data are exchanged between the Simulation Core and the Device Application during a simulation. The simulation data is retrieved by the GUI for display. Upon building new simulations, the GUI starts the Simulation Generator which accesses the LNS network database and also employs the NC2C Translator and the MSDEV compiler to build the new simulation. The simulation configuration is temporally stored and is made accessible to the Simulation Core.

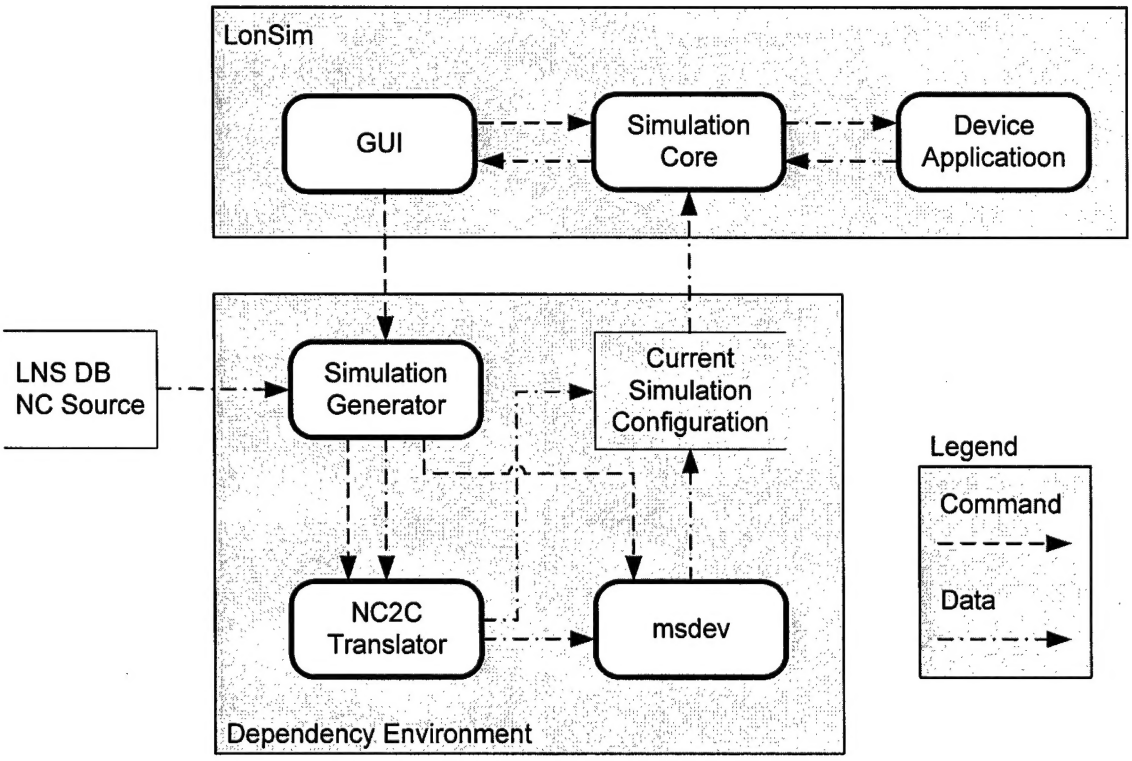


Figure C.1. Software Architecture and Dependency.

APPENDIX IV: LNS Database

The LonWorks Network Service (LNS) is a multi-client network operating system for installing, upgrading, and interacting with LonWorks networks. The LNS operating system employs a server (the LNS server) to maintain various databases (the LNS databases), including a single global database and a several network databases. The LNS global database contains general information about all of the networks created by the user. Detailed information about each network—including device configuration data—is maintained in individual LNS network databases. When a user creates a network design using LonMaker, the global database is updated and a new network database is created to store the network configuration.

LNS provides a common API to access the LNS database for programming LNS COM component applications. The same API can be used to access LNS databases and retrieve network configurations. VDCS makes use of this API to build the network simulation directly based on the network configuration in a LNS database. In other words, VDCS re-creates the network design to be simulated directly from the LNS database. The user only needs to define the network design once—with the familiar LonMaker tool—and can use that single design in both the VDCS simulation environment and with the actual hardware.

The previous version of VDCS required the manual duplication of the LonMaker network design in a different environment. The new version of VDCS automates this process by generating a network simulation based on the LNS database.

APPENDIX V: COM Technology

Component Object Model (COM) technology (e.g., Microsoft 2002) is a specification of a binary interface for components and a set of services provided by operating system libraries to enable modular, object-oriented, customizable and upgradeable distributed applications using a number of programming languages. COM specifies the standards for creating interoperable components. A COM component can be packaged in a module such as a Dynamic Link Library (DLL) or an Executable (EXE) image. COM components use binary interfaces to communicate with other modules or applications. The binary interface makes it possible for COM components to be developed using different programming languages.

We applied COM technology to the VDCS test platform in order to improve software modularity. By adopting COM, changes to an individual module will no longer affect other modules in the test platform as long as the interface definitions remain the same. In fact, the other modules need not be re-compiled. The new modular architecture promises to reduce the complexity of software maintenance and upgrades, which are expected to grow as VDCS evolves to handle larger scale systems.

The use of COM components also enables VDCS to execute various simulation scenarios during a single user session. VDCS creates a DLL with the compiled user-application code for each network simulation scenario. VDCS loads a network scenario DLL at run-time and links it to the overall simulation engine. The users select which scenario they are interested in simulating and VDCS loads the appropriate DLL. VDCS can guide the user to make modifications to device-level application code; it can spawn a compiler to re-build a new simulation object (incorporating the user's application code changes); and it can then dynamically link to the new module in order to execute the simulation and display the results. Before the COM architecture was adopted, users had to terminate the VDCS program every time they made changes to the application code (the entire simulation engine was recompiled).

Rebuilding VDCS using COM technology also provides a mechanism to create standardized interfaces between applications. This feature allows us to integrate VDCS with third-party software packages that simulate physical systems, such as fluid systems (e.g., FLOWMASTER, SIMSMART). Since COM technology also supports different programming languages, future VDCS development could be done in higher-level programming languages thus allowing enable more rapid application development when execution speed is less critical (e.g., use of Visual Basic for GUI development).